USENIX Proceedings of the 1999 USENIX Annual Technical Conference *Monterey, CA, USA, June 1999*

conference proceedings

1999 USENIX Annual Technical Conference

Monterey, California, USA
June 6–11, 1999

Sponsored by
USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $32 for members and $40 for nonmembers.
Outside the U.S.A. and Canada, please add
$18 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

1998 New Orleans	1989 Summer Baltimore
1997 Anaheim	1989 Winter San Diego
1996 San Diego	1988 Summer San Francisco
1995 New Orleans	1988 Winter Dallas
1994 Summer Boston	1987 Summer Phoenix
1994 Winter San Francisco	1987 Winter Washington, D.C.
1993 Summer Cincinnati	1986 Summer Atlanta
1993 Winter San Diego	1986 Winter Denver
1992 Summer San Antonio	1985 Summer Portland
1992 Winter San Francisco	1985 Winter Dallas
1991 Summer Nashville	1984 Summer Salt Lake City
1991 Winter Dallas	1984 Winter Washington, D.C.
1990 Summer Anaheim	1983 Summer Toronto
1990 Winter Washington, D.C.	1983 Winter San Diego

ISBN 1-880446-33-2

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

ACKNOWLEDGMENTS

Program Chair
Avi Rubin, *AT&T Labs—Research*

Program Committee
Charles Antonelli, *CITI, University of Michigan*
Partha Dasgupta, *Arizona State University*
Wu-chi Feng, *Ohio State University*
Robert Gray, *Boulder Labs*
Peter Honeyman, *USENIX Association*
Orran Krieger, *IBM*
Anthony LaMarca, *Xerox PARC*
Darrell Long, *University of California, Santa Cruz*
Gary McGraw, *Reliable Software Technologies*
Yoon-Ho Park, *IBM*
Christopher Small, *Lucent Technologies—Bell Laboratories*
Keith A. Smith, *Harvard University*
Mirjana Spasojevic, *Hewlett-Packard Labs*

Invited Talks Coordinators
Clem Cole, *Compaq Computer Corporation*
John Heidemann, *USC/ISI*

Works-in-Progress Session Coordinator
Keith Smith, *Harvard University*

FREENIX Track Committee
Chair: Jordan Hubbard, *FreeBSD*
David Greenman, *FreeBSD*
John Ioannidis, *AT&T Labs—Research*
Angelos D. Keromytis, *OpenBSD*
Kirk McKusick, *Author & Consultant*
Jason Thorpe, *NetBSD*
Nathan Torkington, *Consultant*
Theodore Ts'o, *MIT*

Extreme Linux Workshop Committee
Chair: Peter Beckman, *Los Alamos National Laboratory*
David S. Greenburg, *Los Alamos National Laboratory*
Jon "maddog" Hall, *Linux International/Compaq Computer Corporation*
Rod Oldehoeft, *Los Alamos National Laboratory*
Matt Welsh, *University of California, Berkeley*

The USENIX Association Staff

Administration
Ellie Young, *Executive Director*
Judy DesHarnais, *Conference Director*
Cynthia Deno, *Marketing Director*
Daniel V. Klein, *Tutorial Coordinator*
Gale Berkowitz, *Deputy Executive Director*

USENIX Proceedings Production
Jane-Ellen Long, *Publications Director*
Jennifer Radtke, *Web Publications Coordinator*
Data Reproductions

USENIX Support Staff
Bleu Castañeda
Cami Edwards
Lana Erlanson
Vanessa Fonseca
Dana Geffner
Toni Veglia

External Reviewers
Guillermo Alvarez
Ahmed Amer
Mohit Aron
Arash Baratloo
Sarr Blumson
Steven Carter
Chuck Cranor
Fred Douglis
Kevin Elphinstone
Marc Fleischmann
Liana Fong
Eran Gabber
Steve Gaede
Alain Gefflaut
Richard Golding
Theodore Haining
Bruce Hillyer
Tim Howes
Christopher Howson
Ayal Itzkovitz
Trent Jaeger
Terence Kelly
David Kormann
Thomas Kroeger
David Lewis
Jochen Liedtke
Dean Long
Kostas Magoutis
Hans C. Masing
CC Michael
Vsevolod Panteleenko
Ted Romer
Dafna Sheinwald
Anthony Sloane
David G. Sullivan
Toby Teorey
Alistair Veitch
John Viega

Contents

1999 USENIX Annual Technical Conference
June 6–11, 1999
Monterey, California, USA

Index of Authors

Preface

Welcome to the last USENIX Technical Conference of this millennium. I believe you will find Monterey to be an enjoyable location for this conference.

On the off chance that January 1, 2000, will not mark the end of civilization as we know it, we have put together an excellent technical program consisting of 23 papers with the latest research results in computing systems. In addition to the technical program, we are continuing the very successful FREENIX track that was introduced last year. In addition, the conference includes 24 tutorials on various topics of interest to this community. To top it all off, there will be another parallel track of distinguished invited speakers.

The program committee received 63 submissions. Each paper was reviewed by four committee members. Some of them were sent to outside experts for review. In all, we utilized 38 external reviewers; their names are listed on p. ii of these proceedings. Once all of the reviews were in, the committee braved a snowstorm to meet at AT&T Labs—Research in Florham Park, New Jersey. There, the committee went through all of the papers and decided on 23 of them. Next the papers were grouped by topic. Finally, the committee met with the invited talks coordinator and the FREENIX chair and all of the events were scheduled.

All of this could not have been possible without the hard work of many people. First of all, I would like to thank the USENIX staff, who ease the burden on program chairs more than any other organization. Next I'd like to thank my entire program committee and all of the external reviewers for all of their hard work. As the liaison to the USENIX board, Peter Honeyman was a tremendous help in selecting program committee members. Thanks to Clem Cole and John Heidemann for an outstanding job with the invited talks track, and finally thanks to Jordan Hubbard and the FREENIX program committee for organizing the FREENIX track.

Aviel D. Rubin
Program Chair

Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers

David Petrou
Carnegie Mellon University
dpetrou@cs.cmu.edu

John W. Milford
NERSC
jwm@csua.berkeley.edu

Garth A. Gibson
Carnegie Mellon University
garth.gibson@cs.cmu.edu

Abstract

We describe extensions to lottery scheduling, a proportional-share resource management algorithm, to provide the performance assurances present in traditional non-real time process schedulers. Lottery scheduling enables flexible control over relative process execution rates with a *ticket* abstraction and provides load insulation among groups of processes using *currencies*. We first show that a straightforward implementation of lottery scheduling does not provide the responsiveness for a mixed interactive and CPU-bound workload offered by the decay usage priority scheduler of the FreeBSD operating system. Moreover, standard lottery scheduling ignores kernel priorities used in the FreeBSD scheduler to reduce kernel lock contention.

In this paper, we show how to use dynamic ticket adjustments to incorporate into a lottery scheduler the specializations present in the FreeBSD scheduler to improve interactive response time and reduce kernel lock contention. We achieve this while maintaining lottery scheduling's flexible control over relative execution rates and load insulation. In spite of added scheduling overhead, the throughput of CPU-bound workloads under our scheduler is within one percent of the FreeBSD scheduler for all but one test. We describe our design, evaluate our implementation, and relate our experience in deploying our *hybrid lottery scheduler* on production machines.

1 Introduction

Lottery scheduling from Waldspurger *et al.* is a recently introduced proportional-share scheduler that enables flexible control over the relative rates at which CPU-bound workloads consume processor time [21]. With a proportional-share scheduler, a user running several CPU-bound jobs, such as those found in scientific environments, can easily control the share of CPU time that each job receives. In time-sharing systems, proportional-share schedulers control the relative rates at which different users can use the processor, enabling load insulation among users. One possible policy forces users with CPU-bound processes to consume CPU time at an equal rate, regardless of the number of processes they own. Such control is particularly useful for Internet Service Providers (ISPs) which often have hundreds of competing users simultaneously logged into one machine. With conventional processor schedulers, it is simple for one user to monopolize the system with her own processes. Considering desktop workstations, another policy allows the console user to consume CPU time at a faster rate than remote users so that the console user's windowing system is responsive. Naturally, this idea applies recursively so that individual users can control the relative rates at which their own processes consume CPU time.

Although proportional-share schedulers such as lottery scheduling have powerful and desirable features, they are not in wide use. We set out in this research to see if there were any technical obstacles to overcome in an implementation of lottery scheduling on conventional time-sharing systems. We began with a straightforward implementation of lottery scheduling on the FreeBSD 2.2.5R operating system. CPU-bound workloads performed as advertised. However, when running batch and interactive workloads together, we experienced poorer responsiveness, or "choppiness," with the interactive applications compared with the same workload under the stock FreeBSD scheduler.

The FreeBSD scheduler has features to dynamically favor specific processes that lottery scheduling lacks. In this work we show a variety of dynamic ticket adjustment strategies that are harmless to the lottery scheduling goals and that allow us to provide comparable specializations to favor specific processes in specific conditions.

Our *hybrid lottery scheduler* includes *kernel priorities* to reduce kernel lock contention and *abbreviated time quanta* to increase responsiveness by preempting processes before their quanta expire. Our primary contribution to achieve responsiveness comparable to the FreeBSD scheduler is with *windowed ticket boost*, a scheme for dynamically identifying interactive processes and boosting their ticket allocation. We make *tickets*, the priority abstraction in lottery scheduling, adaptively serve a dual purpose based on measured process behavior. If a process is using less CPU than has been granted by its ticket allocation, we identify it as currently interactive and give it an apparent boost in ticket allocation to influence scheduling order. We accomplish this without impacting the ability of lottery scheduling to control the relative CPU time used by CPU-bound processes. Further, while lottery scheduling gives users and administrators flexible resource control, it does not easily offer the UNIX `nice` semantics in which a user (system administrator) can downgrade (upgrade) one of its processes relative to all others without downgrading (upgrading) the rest of its processes. We present an approximate emulation of the `nice` semantics. Our hybrid lottery scheduler, which has been continually running on two production servers and one personal machine for 1.5 years, provides comparable responsiveness and throughput relative to the FreeBSD scheduler under the benchmarks we run.

The rest of the paper is organized as follows. In Section 2 we describe both the FreeBSD and lottery schedulers. Section 3 explains our extensions to the lottery scheduler while Section 4 details our implementation. We evaluate our hybrid lottery scheduler and compare it with the FreeBSD scheduler in Section 5. In Section 6 we present our experience in deploying our scheduler on production machines. Section 7 summarizes related work, while Section 8 discusses work that we leave for the future. Finally, Section 9 concludes this paper.

2 Background

A process scheduler has several conflicting goals. The scheduler should ensure that interactive processes are responsive to user input despite not being able to always distinguish these processes from non-interactive processes. Batch processes should be scheduled to maximize throughput despite potential lock contention between such processes. While addressing these goals, the scheduler must ensure that no process starves. In this paper we are not concerned with real-time schedulers [10].

2.1 Scheduling in FreeBSD

FreeBSD [7] is a UNIX operating system for the Intel x86 platform based on UC Berkeley's 4.4BSD-Lite [14] release. FreeBSD's scheduler is a typical decay usage priority scheduler [4] also used in System V [8] and Mach [2]. The scheduler employs a multi-level feedback queue in which processes with equal priority reside on the same runqueue. The scheduler runs processes round-robin from the highest priority non-empty runqueue. Long (100 ms) time slices make TLB and cache state flushing infrequent, imposing minimal overhead on CPU-bound processes. The scheduler favors interactive processes by lowering the priority of processes as they consume CPU time and by preempting processes before their quanta expire if a higher priority sleeping process wakes up. The scheduler prevents starvation by periodically raising the priority of processes that have not recently run. FreeBSD's scheduler also employs higher priorities for processes holding kernel resources. These kernel priorities cause processes to release high-demand kernel resources quickly, reducing the contention for these resources.

The FreeBSD scheduler has several limitations. Hellerstein demonstrates the difficulty in constructing "fair-share" systems based on decay usage schedulers [9]. These fair-share systems [12, 5] dynamically adjust the priorities of running processes to obtain specific processor consumption rates over the long-term via nontrivial and potentially computationally expensive operations. Further, FreeBSD provides only rudimentary inter-user load insulation by limiting the number of simultaneous processes that one user may run, and by terminating processes that accumulate more than a certain amount of processor time. These mechanisms prevent a user from starting many processes that consume CPU time slowly and also from executing processes that consume a lot of CPU time over a long duration.

2.2 Lottery Scheduling

Recently, proportional-share schedulers such as Waldspurger's lottery scheduling have been introduced which strive for instantaneous fairness; that is, making fair scheduling decisions against only the currently runnable set of processes [21]. In lottery scheduling, each process holds a number of *tickets*. The scheduler selects which process to run by picking a ticket from the runnable processes at random and choosing the process that holds this winning ticket. Users can set the ticket ratios among processes to determine the expected ratios that their processes are selected to run. Only runnable processes (not sleeping, stopped, or swapped out) are eligible for this lottery. Hence, if one process is always runnable while another with equal tickets sleeps periodically, the first will consume more CPU time because it has a greater fraction of the system's tickets during the times that the second was sleeping.

Currencies enable *load insulation* among users, making the rate that a user can consume CPU time independent of the number of processes owned. While processes hold tickets in per-user currencies, users hold tickets in a system-wide base currency. For simplicity, per-user currency tickets will be called "tickets," and base currency tickets will be called "base tickets." The ticket distribution within a user's processes determines the relative rate of execution among these processes. To control the distribution of CPU time among users, the system administrator can vary the number of base tickets held by a user. This varies the total execution rate of all the user's processes with respect to processes owned by other users. The scheduler accomplishes this by first converting the tickets, T_p, belonging a process into a number of base tickets, B_p, and then performing a lottery with base tickets instead of tickets. In detail, $B_p = ET_p$, where E, the exchange rate, is the number of base tickets funding a user's currency divided by the total number of tickets across the user's runnable processes.

Lottery scheduling employs *compensation tickets* to enable a process which goes to sleep before exhausting its time quantum its fair share of CPU time if it becomes runnable before the next scheduling decision is made. Each time a process uses only a fraction, f, of its quantum, the scheduler compensates the process by temporarily increasing its tickets until the next time the it is chosen by the scheduler. When compensated, a process holds an *effective* number of tickets equal to $T_p(1/f)$. For example, a process with 10 tickets that yields the CPU after using $1/2$ of a quantum will hold 20 effective tickets, enabling it to be chosen twice as likely as it would be without compensation tickets. This type of technique—dynamic ticket adjustments to achieve specific behavior without sacrificing the overall goals of proportional-share scheduling—is the basis for the mechanisms described in this paper.

3 Hybrid Lottery Scheduler Design

We extended lottery scheduling to be more responsive to interactive applications and to reduce kernel lock contention using techniques borrowed from the FreeBSD scheduler. Further, user feedback prompted us to provide semantics similar to the UNIX `nice` utility. The challenge was to improve performance without squandering the desired features of proportional-share scheduling. We describe our extensions resulting in a *hybrid lottery scheduler* in the following sections.

3.1 Kernel Priorities

Processes that block in the kernel often hold shared kernel resources locked until they wake and leave the kernel, such as when a process holds a buffer locked while it waits for disk I/O to complete. FreeBSD schedules processes asleep in the kernel with a higher priority than processes which exhaust their quanta at user level so that they will release these resources sooner, reducing the chance that other processes will find these resources in use [14]. The FreeBSD scheduler implements this by temporarily assigning static kernel priorities to processes after blocking in the kernel so that they will be preferentially scheduled upon waking. These kernel priorities are ordered in importance of the resource held. For example, a process holding a vnode locked will have a higher priority than a process holding a buffer waiting for disk I/O because vnodes have been deemed a more contended or important resource.

In lottery scheduling, a blocked process could temporarily transfer its tickets to the process that holds the desired resource, encouraging it to run and release the resource sooner. This technique was originally introduced to solve priority inversion [21]. However, it is complex to retrofit a kernel such as FreeBSD to perform ticket transfers at each point where a process may block on a kernel resource because of the number of places where new code would have to be added and validated. We cannot simply interpose ticket transfers within the kernel sleep and wakeup functions, because the arguments to these functions do not encapsulate enough information to know to which process tickets should be transferred. Further, this approach incurs more overhead than the FreeBSD scheduler. A process that needs a locked kernel resource will find the resource in use by another process, transfer its tickets to that process, block to enable a new lottery, and eventually recover its transferred tickets when it wakes up. Instead, by preferentially scheduling processes that wake up holding kernel resources, kernel priorities reduce outright the chance that resources are found locked. This reduces the number of context switches because processes contending for these resources will block less often.

Rather than introduce in-kernel ticket transfers to address kernel resource contention, we preferentially schedule processes holding kernel resources, similar to the FreeBSD approach. In detail, we maintain a list of processes that wake up after being blocked on a kernel resource. This list is sorted by the kernel priority of the resource that each process was blocked on. When making a scheduling decision, we run the standard lottery scheduler algorithm if the list is empty. If not, we choose the first process on this sorted list, emulating the behavior of the FreeBSD scheduler.

We lose the probabilistic fairness of lottery scheduling by choosing processes outside of the standard lottery scheduling algorithm. We recover this fairness with a variation of compensation tickets (see Section 2.2). We track the total time that each process has run between two successive selections by the standard lottery scheduling algorithm. During this period the process may go to sleep several times as it blocks in the kernel. When the process gets descheduled in user space by the time slice interrupt, the scheduler computes its compensation tickets based on this total time. If the process ran longer than one time quantum, an appropriate number of *negative* compensation tickets is temporarily assigned until the process is chosen to run again. These negative compensation tickets make the process *less* likely to be chosen by the scheduler. For example, consider a process with 10 tickets that overran its time quanta by 25%. The scheduler will negatively compensate the process by 2 tickets so that it will hold only 8 effective tickets until the process is chosen by the lottery. In detail, effective tickets $= T_p(1/f) = 10(1/1.25) = 8$.

Sometimes a process spends very little time in user space because it continually makes blocking system calls, making it unlikely that the time slice interrupt will deschedule it. As described, when the process is eventually descheduled by the time slice interrupt, it will receive a large number of negative compensation tickets, causing it to wait a long

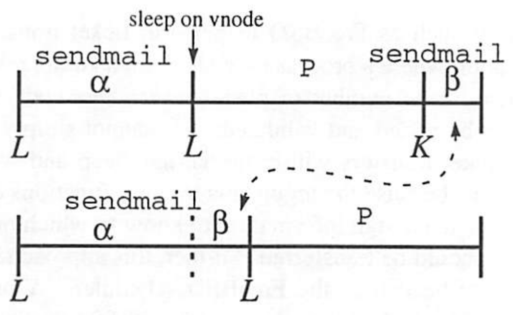


Figure 1: This figure shows processes being scheduled as time moves from left to right. L indicates a scheduling decision by the standard lottery scheduling algorithm while K shows where a process holding a kernel priority runs without having been chosen by the standard lottery scheduling algorithm.

time before running again. To avoid this, we force the process to relinquish the CPU in user level as soon as it has consumed more than one time quantum and is not executing in the kernel. This way the process will incur a small number of negative compensation tickets more frequently.

Figure 1 shows one way to think about kernel priorities in the hybrid lottery scheduler. On the top line `sendmail` is chosen by the lottery scheduler and after α time goes to sleep waiting for locked vnode, perhaps for the mail spool directory. Meanwhile, another process, denoted `P`, gets chosen by the lottery scheduler. When the vnode becomes available, `sendmail` is chosen from the kernel priority list (at K) and runs for some time, β, at which point the time slice interrupt occurs. The hybrid lottery scheduler grants `sendmail` compensation tickets at the end of β exactly as if `sendmail` was able to acquire the vnode lock without blocking and had run contiguously for time $\alpha + \beta$ as shown on the bottom line. The scheduler is in the same state at the end of both lines, showing that kernel priorities only temporarily violate the proportional-share properties of lottery scheduling.

3.2 Abbreviated Quanta

Lottery scheduling succeeds at scheduling processes which never voluntarily relinquish the CPU in proportion to the number of tickets that they hold. Interactive jobs, such as editors and shells, spend most of their time idle, and thus for them we are not concerned with the rate at which they consume CPU time, but instead with how responsive they are to user input. Upon becoming runnable from receiving input, we desire the time it takes for the scheduler to choose the process, known as the *dispatch latency*, to be small. Since lottery scheduling does not distinguish between CPU-bound and interactive jobs, it often fails to schedule interactive jobs first, causing noticeable "choppiness." This and the following sections describe enhancements to lottery scheduling to improve responsiveness.

Under the FreeBSD scheduler, a process waking up after blocking in the kernel (such as when a process waits for an I/O event) will preempt the running process before its quantum expires. The process which woke up has a kernel priority and will be chosen to run next unless multiple such tasks arrive at once. This behavior, which we call *abbreviated quanta*, is desirable if the sleeping process is interactive, for example, an `emacs` process that has just received a keystroke. Further, there are instances in which FreeBSD will *conditionally* preempt the running process. One such instance is when a process receives a signal and has a higher priority than the running process. Lottery scheduling, however, does not preempt the running process when another process wakes up or receives an event[1].

Our hybrid lottery scheduler forces a context switch when a sleeping process wakes up or when a process receives an event such as a signal. To ensure that the preempted process will receive the processor in proportion to the number of tickets that it holds, the scheduler awards the preempted process compensation tickets based on the fraction of the time slice it used (see Section 2.2).

We note a synergy between kernel priorities and abbreviated quanta. Without kernel priorities, abbreviated quanta will preempt running processes when a process wakes up, but will not guarantee that processes that were blocked in the kernel will be chosen ahead of processes executing user-level code. Without abbreviated quanta, kernel priorities will schedule processes blocked on kernel resources ahead of those that are not, but will not preempt processes before their quanta expire.

3.3 Windowed Ticket Boost

Since interactive processes generally use a fraction of their quanta, compensation tickets make them more responsive by temporarily boosting their effective tickets. Unfortunately, compensation tickets cannot be relied on to provide adequate responsiveness. If a CPU-bound job gets preempted by another process due to abbreviated quanta, then it will also receive compensation tickets, putting it on equal footing with interactive processes.

Consider the interactive application `emacs` running in an X window while a CPU-bound job runs in the background. At the outset, X runs while `emacs` sleeps waiting for a keystroke. Earlier, the CPU-bound job was preempted and received compensation tickets. When we press a key, `emacs` preempts X and runs immediately because we have implemented abbreviated quanta and `emacs` holds a kernel priority. After processing the keystroke `emacs` goes back to sleep. Both X and the CPU-bound process are runnable and neither have a kernel priority because they were both preempted running user-level code. Up to this point, both the

[1]Other implementations of lottery scheduling may have implicitly leveraged abbreviated quanta on the operating systems they were implemented on. However, we know of no work demonstrating the importance of using abbreviated quanta with lottery scheduling.

FreeBSD and hybrid lottery schedulers behave the same. The FreeBSD scheduler will almost certainly pick X to run because, having accrued less CPU time than the CPU-bound process (which always runs when X and `emacs` are sleeping), it resides in a higher priority runqueue. Because neither X nor the CPU-bound process have kernel priorities, the lottery scheduler will perform the standard lottery algorithm against them. Sometimes the CPU-bound process will be chosen and run for one quantum (100 ms). When this occurs, X delays in updating the `emacs` window, resulting in choppiness.

We have experienced this scenario on a workstation running a preliminary version of our hybrid lottery scheduler with only abbreviated quanta and kernel priorities implemented. When holding a key down (equivalent to pressing 120 keys/s) over approximately 7 s, we generated 825 key events, of which 23 where delayed when the CPU-bound process was chosen to run before X, perceptibly delaying window updates. In summary, delayed dispatching of an interactive process is likely to happen if it does not hold a kernel priority (because it was preempted at user level) and a CPU-bound process was preempted and granted compensation tickets. This will also occur in deterministic proportional-share schedulers like stride scheduling.

To solve this problem we make tickets serve a dual purpose depending on the nature of the process. For CPU-bound jobs, tickets correspond to the rate of CPU time consumed, as before. For interactive jobs, tickets determine how responsive the processes are. If we increase the tickets held by interactive jobs, then we increase the chance that they will be chosen to run before CPU-bound jobs, shortening dispatch latency. Hence, if we can identify processes as interactive, we can boost their tickets to make them more responsive.

Our central observation is that most interactive processes do not use CPU time in proportion to their ticket allocations because they often block for long time periods on user input. By tracking the total number of base tickets in the system and total number of tickets per currency at regular intervals, we can find the processes that have received less CPU time than deserved. Transparently to the user, we classify these processes as interactive and boost their effective tickets to make them more responsive. There is no fear that they will use more than their allotted share because we only boost the tickets of exactly those processes using less than their share.

We desire the dispatch latency of interactive jobs to be below the minimum latency that humans can discern. When only one interactive job is runnable, abbreviated quanta and an arbitrarily high ticket boost will ensure this. However, when multiple interactive jobs are simultaneously runnable, there is a chance that the dispatch latency of some of them will be noticeable. We can reduce this chance by influencing the scheduling order of multiple interactive jobs to minimize their average dispatch latency. It is well known that response time is minimized by scheduling tasks via shortest-processing-time first (SPT) [3]. If we knew how much processing time would be used by the interactive jobs in handling an event (such as a keystroke) before going back to sleep, we could schedule them optimally to minimize dispatch latency. Since we do not have this information, we approximate SPT by using history. We make the ticket boost inversely related to the fraction of deserved CPU time used. Specifically, we boost the effective tickets of a job that used almost none of its allotted CPU time by a large factor, such as 10,000, making it many times more likely to be chosen as before. A job that used its deserved share receives no boost. Finally, we use interpolation to derive the boost of a job whose fraction of deserved CPU time was between these extremes. Hence, interactive jobs that barely use the CPU are more likely to be scheduled before interactive jobs that use a moderate, but still less than deserved, amount of CPU, approximating SPT.

Jobs are rarely continuously interactive or CPU-bound. For example, an `emacs` process may be interactive most of the time, but then become CPU-bound for some time as it executes elisp code. To adapt to these types of processes, we look at a sliding window interval of history when checking to see if a process is using its fair share of the CPU. We would like the scheduler to be "agile," in the sense that if a previously interactive job became CPU-bound, we restore its ticket allocation quickly, and contrariwise, if a previously CPU-bound job became interactive, we quickly boost its ticket allocation. This argues for a small window size, *e.g.,* on the order of one second. However, if the window size is too small, there is a chance that a CPU-bound job will not run during this window simply because the machine is heavily loaded. If this were to occur, we would mistakenly label such a process as interactive. While this parameter may require tuning for different environments, we found that a window interval of ten seconds works adequately for making a bimodal process such as `emacs` responsive within a few seconds after going from a CPU-bound to interactive stage, when competing against completely CPU-bound processes.

We call this extension *windowed ticket boost.* Our motivation was the priority decay present in the stock FreeBSD scheduler. Rather than lower the priority of processes consuming CPU time, we raise the priority of those that do not. Since we only boost the priorities of only those processes that are not receiving their deserved CPU time, we do not violate the proportional-share semantics. The combination of the abbreviated quanta and windowed ticket boost extensions together provide the responsiveness of the stock FreeBSD scheduler.

3.4 `nice` Emulation

All UNIX variants have a utility called `nice` that enables a user to vary the priority of her processes from -20 (highest) to $+20$ (lowest) relative to all other processes in the system. After deploying our hybrid lottery scheduler on production systems, users complained that ticket adjustments alone did not give them the type of control provided by `nice`. Since `nice` priority adjustments only allow a general favoring or shunning of process priority and do not provide absolute, or even predictable guarantees [18], we do not attempt to exactly emulate `nice` semantics, but rather we approximate the aspects of its behavior relied on by its users; specifically, that a user (system administrator) can downgrade (upgrade) one of her processes relative to all others without downgrading (upgrading) the rest of her processes. Of course, the targeted process will no longer receive the CPU utilization allocated by its tickets.

A naïve approach to emulating `nice` semantics is to only modify the tickets of the target process according to a mapping of `nice` values to tickets. This fails because only the relative execution rate of a user's own processes are affected. If a user has only one CPU-bound process running that is `nice`'d to lowest priority ($+20$), the exchange rate will still assign the process all of the base tickets funding the user's processes (see Section 2.2). Another incorrect approach adjusts only the base tickets funding a user's processes with a mapping of `nice` values to base tickets. In this instance, if a user lowers the priority of one process with `nice`, all of the user's processes will have lower priority relative to the rest of the processes on the system. One could instead create a new currency for each `nice`'d process, funded with a number of base tickets reflecting the `nice` value. Unfortunately, a user can run a large number of `nice`'d processes and get an unfair share of CPU time. We could not take that many base tickets from her normal currency, because then her normal processes would not get their share of CPU time if any `nice`'d process momentarily went to sleep. Further, without adding more complexity, we would run out of base tickets to take from the user's currency if the user started many `nice`'d processes.

Our solution adjusts both tickets and base tickets to achieve the `nice` function. We raise (lower) a process's number of tickets according to the process's `nice` value, ensuring that the process will have more (less) priority relative to other processes owned by the user. When a scheduling decision is made, we convert the tickets held by the process into base tickets and make the following adjustments to the allocated base tickets. If the process has been `nice`'d to have a lower priority, we force the number of base tickets held by the process to be *at most* a threshold determined by the `nice` value. Likewise, if the process has been assigned a higher priority with `nice`, we force the number of base tickets held to be *at least* a specific threshold. This thresholding ensures that a `nice`'d process will have more or less priority relative to processes outside of the user's currency.

4 Implementation

Our system is divided into two parts. Most of the code resides in the kernel files implementing the hybrid lottery scheduler. The rest of the system consists of small user-level programs that make a new *lott_sched()* system call to adjust or query scheduling parameters.

4.1 Kernel Functionality

We first describe critical pieces of the FreeBSD scheduler necessary for understanding our scheduler implementation and for understanding the benchmarks in Section 5. After becoming runnable, processes are put onto the appropriate runqueue by the assembly language routine *setrunqueue()*. Processes are removed from runqueues when chosen by the scheduler. The assembly language routine *cpu_switch()* saves process context, chooses the next process to run, and switches to that process (or idles if no processes are runnable).

We decoupled the scheduling policy from mechanism by placing a function call from within *cpu_switch()* to *lott_choose_next_runner()*, a C function which implements the hybrid lottery scheduling algorithm. We assign processes a default number of tickets when they are created by *fork1()*. The *setuid()* system call (*cf.*, `login`) was modified to create a new user currency funded with a default number of base tickets. Naturally, we do not create a new currency if the user is already logged onto the machine. We reference count the processes owned by a user so that we can garbage collect her currency when all of her processes terminate. Although Waldspurger's original framework allows currencies to be nested indefinitely, our implementation only distinguishes between the base and per-user currencies.

Any lottery scheduler with compensation tickets and currencies will be more computationally expensive than the FreeBSD scheduler. A FreeBSD scheduling decision is a fast $O(1)$ operation because the scheduler simply removes a process from the head of the first non-empty runqueue. Our hybrid lottery scheduler implementation employs an $O(n)$ algorithm, where n is the number of runnable processes[2]. As we sought to achieve performance comparable to the FreeBSD scheduler, we expended much effort in optimizing our implementation. Since floating point operations are not permitted while running in the FreeBSD kernel, we do nearly all of our computations with fixed-point integers. We use 32-bit integers so that we can utilize hardware instructions for most of the arithmetic. When we needed to use 64-bit integers, we wrote an in-line assembly routine which issues the "32-bit times 32-bit to 64-bit" hardware instruction after discovering that `gcc` inefficiently compiled this operation. We also defer work, use in-line

[2]An $O(\lg n)$ lottery scheduling algorithm exists, but as we rarely see large n, we believe its extra overhead outweighs its lower computational complexity.

```
lott_choose_next_runner() {
    if(empty(kernel_pri_list && lottery_list))
        return 0; /* idle */
    if(!empty(kernel_pri_list))
        return head(kernel_pri_list);
    foreach(process on lottery_list) {
        compute effective tickets based on
          ticket_boost or frac. of quanta used;
        convert effective tickets to base tickets
          based on user's exchange rate;
        use threshold if process is nice'd;
        update our running count of base tickets;
    }
    pick random number from 1 to total base
      tickets;
    foreach(process on lottery_list)
        if(process holds winning base ticket)
            return process;
}
```

Figure 2: Pseudo-code for *lott_choose_next_runner()*.

functions, and aggressively cache the computed values described further below.

While in theory the number of tickets or base tickets allocated to a process or user, respectively, can be any positive integer, implementation efficiency encourages the range of values to be bounded. We bound the ratio of minimum to maximum ticket and base ticket shares from 1–100 as we believe two orders of magnitude expresses sufficient resolution. We carefully set the range of tickets and base tickets from 1–100 and 100–10,000 respectively so that we would not incur overflows and underflows during our computations with 32-bit fixed-point integers. By default, processes and users start with 100 tickets and 1,000 base tickets respectively.

To help us optimize and understand the system, we instrumented both the FreeBSD and our hybrid lottery schedulers to provide us with detailed profiling information. We can measure the last n scheduling events, including when and for what reasons processes go to sleep, and time quanta expirations.

To implement windowed ticket boost, we maintain circular arrays that track whether a process is getting less than its share of CPU time. Every second we update a circular array that holds the number of base tickets in the system over the last ten seconds. We also update similar arrays which track the number of tickets held in each user's currency. Updating the circular arrays more often would give us more accurate data at the expense of higher overhead. For a particular process, *setrunqueue()* derives the amount of time it deserved to run from the values in the circular arrays. Then this function finds the fraction, f, of deserved time actually used over the ten second interval. If f is greater than one, the process is getting at least its fair share of the CPU. If not, we deem the process interactive and derive a scaling factor called `ticket_boost` which we later multiply against the process's effective tickets. While `ticket_boost` can simply be set to $1/f$, raising this to a power will dramatically increase our system's ability to ensure SPT ordering for interactive jobs while still using a probabilistic scheduler. Although this parameter may require tuning for different environments, our current implementation uses $(1/f)^4$. To prevent an overflow in a later computation, we clip `ticket_boost` at 10,000.

If a process has been asleep for more than ten seconds, we discount it from the above measurements because it is not actively competing for CPU time. When such a process wakes up, *setrunqueue()* assigns it the maximum `ticket_boost` since it most probably has not received its share of CPU time.

The heart of the hybrid lottery scheduling algorithm resides in *lott_choose_next_runner()*, illustrated in Figure 2. The lists `kern_pri_list` and `lottery_list` contain runnable processes that do and do not hold kernel priorities, respectively. If a process used less CPU time than it deserved, we compute the process's effective tickets by multiplying the number of tickets that the process holds by its `ticket_boost`. If not, we are dealing with a CPU-bound process and need to assign compensation tickets. We compute its effective tickets by dividing the number of microseconds in one time quantum by the number of microseconds used by the process during the last time that it was scheduled and multiplying this value by the number of tickets that the process holds. We compute the base tickets held by the process by dividing the number of base tickets that fund the user's currency by the number of tickets in all of the user's runnable processes and multiplying this exchange rate by the process's effective tickets.

When a process is `nice`'d, we change the number of tickets that it holds so that it will have more or less priority relative to the rest of the user's processes. Since tickets range from 1 to 100, a given `nice` value equals $10^{\frac{1}{20}(-\texttt{nice}+20)}$ tickets. After converting a process's tickets to base tickets in *lott_choose_next_runner()*, we apply the `nice` base ticket threshold. Since base tickets range from 100 to 10,000, the mapping from `nice` values to base tickets is $10^{\frac{1}{20}(-\texttt{nice}+20)+2}$. If the process has been `nice`'d to have a lower priority, we force the number of base tickets held by the process to be *at most* this value, and vice-versa if the process has been assigned a higher priority with `nice`.

4.2 User-level Programs

Users adjust and query lottery scheduling parameters via a number of user-level programs. `set_tickets` changes the number of tickets held by a process while `show_tickets` displays a `ps`-like listing of a user's processes and their ticket allocations. A user executes a given program with a specific number of tickets with `run_tickets`. Root uses `set_funding` to set the number of base tickets that fund a

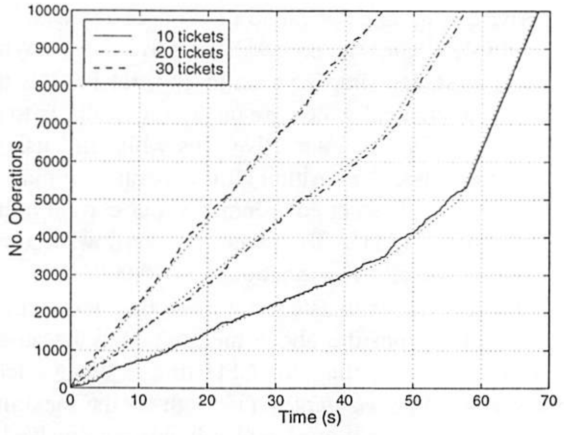


Figure 3: This figure demonstrates the progress of three CPU-bound processes under the hybrid lottery scheduler. Each operation on the Y-axis represents a fixed amount of work. Also shown are straight dotted lines representing ideal processor utilization. Notice that when a process finishes, after completing 10,000 operations, the remaining processes execute faster.

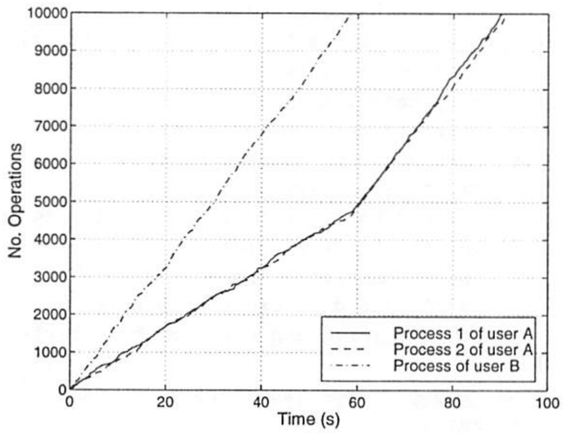


Figure 4: This figure shows the progress of three CPU-bound processes under lottery scheduling. The top curve represents a process run by one user while the bottom curves represent two processes run by another user. When the first process finishes, the other processes have made 47% and 46% progress toward completion (50% each is ideal).

user's currency while `show_funding` displays the number of base tickets that fund a user's currency. Today the funding commands only apply to logged in users; soon we will record the number of base tickets funding a user's currency in the user's account record. Finally, `lott_chuser` takes a process and places it under another user's currency, which is useful for moving processes like the X Window System which run as root under the currency of the process's primary user.

5 Evaluation

We first demonstrate some properties of proportional-share resource management that the FreeBSD scheduler lacks. We then show that the hybrid lottery scheduler is more responsive and helps reduce waiting time for processes blocked on kernel resources relative to our initial lottery scheduler implementation. Finally, we show that the hybrid lottery scheduler incurs minimal overhead relative to the FreeBSD scheduler.

Unless otherwise noted, all results in this section were obtained from the personal computer called `partita`. This machine has one 200 MHz AMD K6 (Pentium compatible) processor, 64 MB of main memory, and 3 GB of ultra-wide SCSI storage. No tests caused the machine to page.

5.1 Flexible Execution Rate Control

Here we demonstrate the features that we have gained by replacing FreeBSD's decay usage scheduler with lottery scheduling. We refer the reader to Waldspurger's thesis [20] for a wide range of additional examples and an extensive analysis of lottery scheduling.

We demonstrate the ease with which a user can control the execution rate of her programs in Figure 3. This figure shows three CPU-bound processes assigned tickets in a 3:2:1 ratio making progress at approximately the same ratio. Curious as to how hard this is to achieve using the FreeBSD scheduler, we found through trial and error a number of `nice` values that come close. The `nice` values that we discovered, +10, +5, and 0, are not intuitively mappable to our goal of 3:2:1. Naturally, any other ratio would be equally difficult to implement. Further, while lottery scheduling maintains the 3:2:1 ratio irrespective of system load, the FreeBSD scheduler unpredictably schedules these processes if other jobs are running.

We demonstrate user workload insulation in Figure 4. Despite one user running two CPU-bound processes, the second user is able to receive approximately twice the throughput from one CPU-bound process.

5.2 Performance

In this section we show the utility of windowed ticket boost, abbreviated quanta, and kernel priorities.

The minimum latency that humans can discern varies between 50–150 ms depending on the individual [15]. We set up the following experiment to test the responsiveness of our hybrid lottery scheduler. We run one completely CPU-bound process against a bimodal process, both with the same number of tickets. This bimodal process is CPU-bound for the first 15 seconds, interactive for the next 20 seconds, and then CPU-bound again for the remaining 15 seconds. In the interactive stage, the process repeatedly computes for approximately 5 ms and then sleeps for about 100 ms, simulating an editor such as `emacs`. While these

	Dispatch Latency (milliseconds)			
	0–15 s	15–35 s	35–50 s	22–35 s
Without Windowed Ticket Boost	29.85 (58.18)	9.13 (27.79)	27.24 (49.64)	10.59 (31.63)
With Windowed Ticket Boost	27.79 (58.46)	2.68 (12.84)	27.58 (50.92)	0.05 (0.02)

Table 1: This table presents the data from Figures 5 and 6 numerically. We show the average dispatch latency (and standard deviation in parentheses) in milliseconds for the bimodal application for each of its three stages of execution, with and without windowed ticket boost. Between 15–35 seconds we perform better with windowed ticket boost, although some of this time is spent discovering that the process became interactive. In the fourth column we show the dispatch latency during seconds 22–35, which is when the scheduler has identified the process as interactive and is applying a substantial ticket boost.

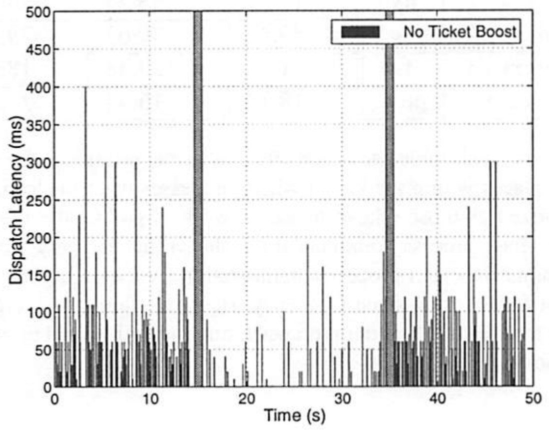


Figure 5: This figure shows the dispatch latency of a bimodal process competing against a CPU-bound process under our hybrid lottery scheduler with windowed ticket boost disabled. From seconds 15 to 35 (marked by the thick bars) the process is interactive, yet the scheduler is unable to schedule it in a timely manner.

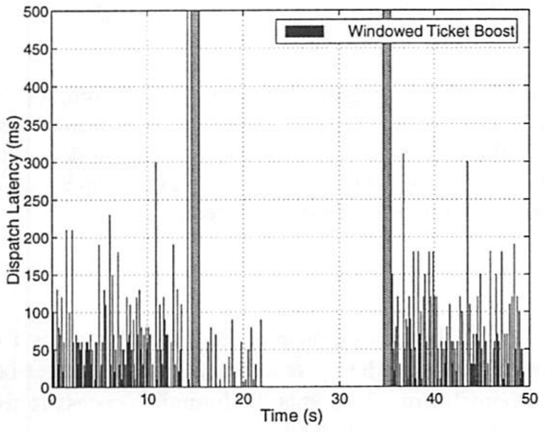


Figure 6: This figure shows the dispatch latency of a bimodal process competing against a CPU-bound process under our hybrid lottery scheduler with windowed ticket boost enabled. From seconds 15 to 35 (marked by the thick bars) the process is interactive and after a short adjustment period, the process exhibits excellent dispatch latencies.

two processes compete, a third process wakes up every 50ms and goes back to sleep immediately simply to cause occasional rescheduling events as would be triggered by X or background processes in an actual workload.

The most important metric for responsiveness is dispatch latency after user input, that is, the time elapsed between when a process becomes runnable in *setrunqueue()* and when it is chosen to run by *lott_choose_next_runner()*. We ran the described benchmark with and without windowed ticket boost as shown in Figures 5 and 6, and Table 1. During the interactive phase (seconds 15–35) the scheduler with windowed ticket boost disabled is unable to schedule the process fast enough to avoid discernible choppiness. Although the dispatch latency is somewhat lower than during the CPU phases due to compensation tickets, there are still many points over 100ms. This occurs because the CPU-bound process occasionally gets preempted, earning compensation tickets which puts it on equal footing with the interactive process. When windowed ticket boost is enabled, we see an adjustment period for about seven seconds after the bimodal process becomes interactive. As the 10 second sliding window moves forward, the process's `ticket_boost` increases until it is always favored by the scheduler when it becomes runnable. From seconds 22 to 35, the dispatch latency is in the tens of microseconds, far below human perception. Once the process becomes CPU-bound again, the system quickly adapts by lowering the process's `ticket_boost` parameter, ensuring that it will not get more than its share of CPU time.

To ensure that windowed ticket boost does not negatively effect the completely CPU-bound process, we show the progress of both processes with windowed ticket boost enabled in Figure 7. Between seconds 0–15 both processes consume CPU time at the same rate. When the bimodal process becomes interactive between seconds 15–35, the CPU-bound job nearly doubles its throughput, getting at least its share. Finally, when the interactive process becomes CPU-bound from seconds 35–50, the sliding window quickly adapts to lower its `ticket_boost` so that it does not dominate the CPU. The bimodal job only needs to consume the CPU at a faster rate for less than one second before losing its boost. We omit the same graph with windowed ticket boost disabled because it is nearly identical.

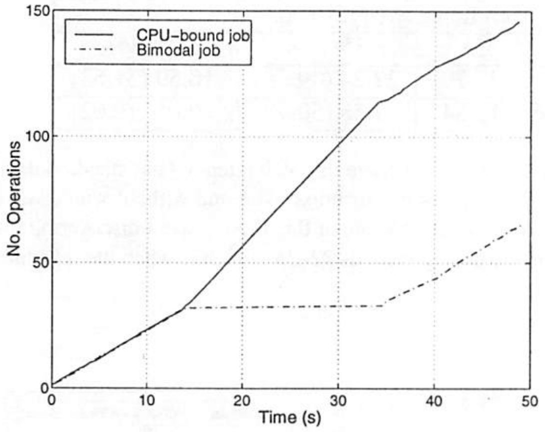


Figure 7: This figure shows the computational progress of the CPU-bound and bimodal jobs with windowed ticket boost enabled. When the bimodal job enters its interactive stage between seconds 15 to 35, the CPU-bound job makes progress 1.9 times faster than when both jobs are CPU-bound.

		% improvement with kernel priorities		
wmesg	type	freq.	duration	weighted
biowait	disk	4.43	20.38	23.91
ffsfsn	fs	8.53	25.26	31.63
getblk	fs	39.37	39.15	63.11
pipdwt	ipc	75.97	27.33	82.54
piperd	ipc	44.51	−1487.28	−780.72
sbwait	net	−14.11	63.39	58.22
swpfre	vm	−101.97	52.75	4.57
swread	vm	16.09	34.53	45.06
ttywai	tty	−9.25	53.08	48.74
ttywri	tty	11.69	53.74	59.15
ufslk2	fs	59.43	49.63	79.57
vnread	fs	−11.62	27.38	18.94
wait	proc	18.12	15.94	31.18

Table 2: This table shows the effect of kernel priorities. We show the places within the kernel where processes went to sleep, categorized as those related to the network, disk I/O, the file system, inter-process communication, the virtual memory system, terminal I/O, and process administration. We compare the percent reduction in sleep frequency, sleep duration, and weighted (by frequency) sleep duration when running with kernel priorities enabled.

Without abbreviated quanta, the dispatch latency for the bimodal process when it is interactive would never be below about 100ms. Note that the bimodal process is always chosen by the scheduler at the start of a new quantum. After it goes to sleep, the scheduler chooses the CPU-bound process. Unless abbreviated quanta is employed, it will consume CPU time until the time slice elapses.

To show the utility of kernel priorities, we instrumented the FreeBSD kernel to provide us with statistics concerning kernel lock contention. Every 15 minutes we took a 30 second log of each time a process went to sleep (via *tsleep()*), for what purpose it went to sleep, and for how long. We ran one set of numbers with kernel priorities enabled, and one set without, each for a 24-hour period on soda, a busy production machine on which we have deployed our hybrid lottery scheduler. Although there is some uncertainty in our measurements due to our inability to completely control the workloads over both runs, we took care to ensure that the workloads were roughly equivalent by comparing the number of users logged in, the context switch rate, and the paging activity.

Over a 24-hour period, there were about 40 distinct reasons why processes went to sleep. We present an abbreviated version in Table 2. We omit sleeps that occur less than 100 times and sleeps initiated by processes that neither held a lock before going to sleep nor held one upon waking (such as when a process goes to sleep on a timer event). We compare the percent reduction in sleep frequency, sleep duration, and weighted (by frequency) sleep duration when running with kernel priorities enabled. Due to latency variations in network and terminal-related events which cause long-tailed wait distributions, we used the sleep duration median to generate the data presented.

With kernel priorities, processes holding kernel resources are preferentially scheduled, reducing the duration of sleeps, which in turn reduces the frequency of sleeps because there is a smaller window of time that a process will find a resource in use. These trends are apparent although we made no effort to artificially increase kernel resource contention. One major anomaly is the wait duration for processes going to sleep waiting for data in a pipe read (piperd). In the run with kernel priorities disabled, we saw an unusually large number of very short sleeps from the process ssh (a secure telnet shell) on a pipe read during one 30 second interval. We believe that this activity caused this anomaly.

5.3 Overhead

We measure scheduling code fragments to quantify scheduling overhead. To obtain accurate measurements, we employ the RDTSC (Read Time-Stamp Counter) instruction which reads a counter incremented every clock cycle. In the following figures, error bars represent 95% confidence intervals. The number of independent runs for each experiment is listed with the experiment.

The two most common scheduling operations in both the FreeBSD and hybrid lottery schedulers are *cpu_switch()* and *setrunqueue()*. Under the FreeBSD scheduler, *cpu_-switch()* makes a scheduling decision and performs a context switch. In the hybrid lottery scheduler, *cpu_switch()* performs a context switch after calling *lott_choose_next_-*

		FreeBSD		Lottery	
		mean	std. err.	mean	std. err.
1 process	*cpu_switch()*	2.86	0.011	7.25	0.019
	setrunqueue()	0.57	0.003	17.36	0.038
25 processes	*cpu_switch()*	4.18	0.012	14.37	0.124
	setrunqueue()	0.66	0.003	17.79	0.037
50 processes	*cpu_switch()*	4.32	0.012	22.95	0.172
	setrunqueue()	0.64	0.004	17.59	0.062
75 processes	*cpu_switch()*	4.74	0.014	26.63	0.153
	setrunqueue()	0.83	0.003	17.08	0.081
100 processes	*cpu_switch()*	7.37	0.066	36.28	0.241
	setrunqueue()	0.69	0.009	16.63	0.102

Table 3: This table presents the data from Figures 8 and 9 in numerical format. The times are in microseconds. *cpu_switch()* makes a scheduling decision and performs a context switch. *setrunqueue()* marks a process as runnable, and in the hybrid lottery scheduler also computes `ticket_boost` for the windowed ticket boost.

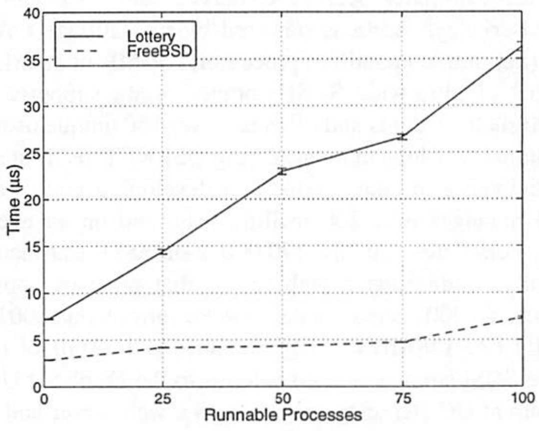


Figure 8: This figure shows the average number of microseconds (out of at least 1,000 measurements) to perform a context switch via the *cpu_switch()* function while varying the number of runnable processes.

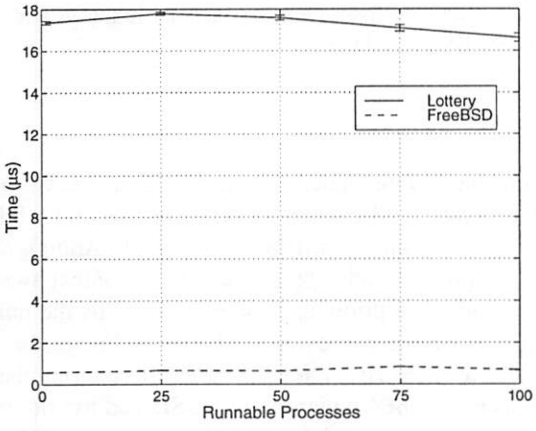


Figure 9: This figure shows the average number of microseconds (out of at least 1,000 measurements) to make a process runnable via the *setrunqueue()* function while varying the number of runnable processes.

runner() which makes a scheduling decision. Figure 8 shows the time it takes to run *cpu_switch()*[3] while varying the number of runnable processes. Naturally, we include the time in *lott_choose_next_runner()* in the hybrid lottery scheduler measurements. As described in Section 4, our scheduling algorithm is $O(n)$ in the number of runnable processes while the FreeBSD scheduler is $O(1)$. This difference in algorithmic complexity is apparent in these results.

Figure 9 shows the time it takes to execute *setrunqueue()*, which makes a process runnable. This function is short in the FreeBSD scheduler. In the hybrid lottery scheduler, we also compute the process's `ticket_boost`, which requires iterating through a 10 element array. While the overhead is substantially higher in the lottery scheduler, this function is $O(1)$ in both schedulers. We do not know why neither curve is entirely flat. Table 3 presents the data from both Figures 8 and 9 in numerical format.

The preceding experiments uncovered measurable differences between the FreeBSD and hybrid lottery schedulers. Now we determine how appreciable these differences are on the scale of compute-bound applications.

To measure the throughput of batch processes we use `rc564`, a program that tries to find the solution to RSA's 64-bit secret-key challenge. To exacerbate the effect of our added overhead while running `rc564`, we increase the number of context switches that occur by running up to 10 processes called `interactive` at the same time. An `interactive` process continually goes to sleep for the shortest time possible and causes a context switch upon

[3]Other work often records context switch time as the elapsed time between passing control from user space in one process to user space in another. Thus our reported times, which record just the elapsed time of *cpu_switch()*, may appear low.

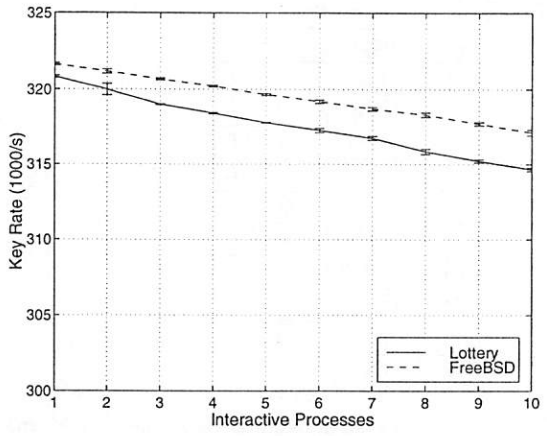


Figure 10: This figure shows the average number of keys tried per second (out of 5 trials) by `rc564` while varying the number of `interactive` processes. Note that the Y-axis begins at 300,000keys/s, exaggerating the apparent differences. The performance under the hybrid lottery scheduler is always within 1% of the FreeBSD scheduler.

waking due to abbreviated quanta. One `interactive` process generates 128 context switches per second while 10 generate 626 context switches per second. Adding an interactive process adds approximately 55 context switches per second. The throughput of `rc564` versus the number of `interactive` processes is shown in Figure 10. We note that as more `interactive` processes are run, the performance of `rc564` under the FreeBSD and hybrid lottery schedulers worsens and diverges. In all runs, `rc564` under the hybrid lottery scheduler is less than one percent slower than under the FreeBSD scheduler.

Curious as to what the context switch rate is on busy systems, we measured `wcarchive`, the world's largest and busiest FTP site[4]. The average number of context switches over a 30 second interval on this site was 2589 per second. As the previous experiment did not show a large difference between the FreeBSD and hybrid lottery schedulers, we ran a program which simply loops and maintains a counter of how many loops it made for 5 minutes, while simultaneously running 100 `interactive` processes. These `interactive` processes pushed the number of context switches per second up to 5160 averaged over the run. In this very extreme test we were about 15% slower than the FreeBSD scheduler. If such an scenario realistically occurred, we could minimize our overhead at the cost of some accuracy by not computing `ticket_boost` on every call to *setrunqueue()*, but perhaps every 10th call.

[4]When we took this measurement in December 1997, `wcarchive` stored 142GB on-line and supported up to and often reached 2750 simultaneous connections. `wcarchive` is located at ftp://ftp.cdrom.com/.

```
PID USERNAME PRI NICE SIZE    RES STAT TIME    WCPU    CPU COMMND
555 jwm       92   0   808K  164K RUN  0:17 16.34% 16.25% rc564
553 peterm    90   0  7392K 8012K RUN  0:18 16.28% 16.21% xoopic
552 peterm    90   0  7392K 8012K RUN  0:18 16.12% 16.06% xoopic
550 peterm    90   0  7392K 7852K RUN  0:18 16.12% 16.06% xoopic
551 peterm    90   0  7392K 7864K RUN  0:18 16.08% 16.02% xoopic
554 peterm    89   0  7392K 8012K RUN  0:18 16.05% 15.98% xoopic
```

Table 4: This table shows the output from `top` while two users are running one and five CPU-bound processes respectively under the FreeBSD scheduler. The lack of load insulation enables `peterm` to obtain an unfair percentage of the CPU.

6 Experience

We have deployed our hybrid lottery scheduler on `soda.-csua.berkeley.edu` and `meeko.eecs.berkeley.edu`, two production machines. `soda` is the central machine for the Computer Science Undergraduate Association at UC Berkeley. `soda` is powered by one 200MHz AMD K6 (Pentium compatible) processor, 256MB of RAM, and 15GB of ultra-wide SCSI storage. `soda` supports over 2400 shell accounts and often has over 150 unique users simultaneously logged on accessing USENET, reading mail, participating in chat rooms, and developing code. `soda` also manages over 200 mailing lists, and on an average day, completes roughly 170,000 `sendmail` transactions. Finally, `soda` runs a web server that receives approximately 50,000 accesses a day. `meeko` runs on one 200MHz AMD K6, 128MB of main memory, and 22GB of ultra-wide SCSI storage. `meeko` belongs to the FreeBSD Users' Group at UC Berkeley. `meeko` runs a web server and mirrors part of `wcarchive` which is offered on its anonymous FTP server. In addition, `meeko` exports a filesystem via NFS. There are usually 5 users logged onto `meeko` actively developing code while a couple dozen users engage in a multi-user game (MUD). These systems have been running our hybrid lottery scheduler since December 1997 (1.5 years). That we have received no complaints is a testament to our implementation's stability and performance.

It is especially important to have load insulation on a machine like `soda` that supports a large user community on one processor. We show the load insulation properties of both the FreeBSD and hybrid lottery schedulers by looking at the output of the UNIX `top` utility while two users run the CPU-bound processes `xoopic` and `rc564`. `xoopic` is a particle-in-cell plasma simulation that calculates fields on a 2-D mesh using Maxwell's equations. Tables 4 and 5 show no load insulation under the FreeBSD scheduler, and reasonably accurate load insulation under the hybrid lottery scheduler.

Our latest version of the hybrid lottery scheduler incorporating windowed ticket boost has not been deployed to `soda` and `meeko` because they are used exclusively over networks of significant latency and thus would not appreciate the benefits offered by this extension.

```
PID USERNAME PRI NICE SIZE   RES STAT TIME   WCPU    CPU COMMND
296 jwm       98   0   808K  392K RUN  0:28 52.21% 48.71% rc564
272 peterm    76   0  7392K 7544K RUN  1:02 11.63% 11.63% xoopic
275 peterm    65   0  7392K 7716K RUN  0:57  9.61%  9.61% xoopic
282 peterm    64   0  7392K 8032K RUN  0:50  9.50%  9.50% xoopic
274 peterm    55   0  7392K 7636K RUN  0:57  7.90%  7.90% xoopic
273 peterm    53   0  7392K 7600K RUN  0:55  7.13%  7.13% xoopic
```

Table 5: This table shows the output from `top` while two users are running one and five CPU-bound processes respectively under the hybrid lottery scheduler. `jwm` is able to receive about 50% of the CPU despite having only one runnable process.

7 Related Work

Process scheduling on time-sharing systems has been studied extensively [13, 11]. A number of fair-share schedulers fairly allocate CPU time to classes of processes over long time spans [12, 5]. Recently introduced proportional-share schedulers such as lottery scheduling [21] and EEVDF [17] strive for instantaneous fairness; that is, making fair scheduling decisions against only the currently runnable set of processes. Another proportional-share scheduler from Waldspurger *et al.* is stride scheduling, which deterministically schedules processes with higher throughput accuracy and lower response time variability compared to lottery scheduling [22]. Since they both employ the same ticket framework, our extensions to lottery scheduling are also applicable to stride scheduling. We choose to extend lottery scheduling over other schedulers because the core algorithm is simple.

Although tickets enable flexible resource control, it is often difficult for users to assign tickets among workloads to meet higher-level performance goals. Recent work from Sullivan *et al.* introduces application-specific "negotiators" that enable automatic ticket exchanges between processes desiring different resource allocations [19]. In other work, a feedback-driven reservation-based scheduler by Steere *et al.* monitors process progress to divine appropriate CPU-time allocations transparently to the user [16].

Arpaci-Dusseau *et al.* studied stride scheduling in the network of workstations context [1]. Part of their aim was to provide better responsiveness under mixed workloads. They award a sleeping (interactive) process exhaustible tickets that expire when it receives its fair share of CPU time. However, most interactive processes will never use their allocation because they are usually sleeping. For these processes, rather than strive for CPU-time fairness, we believe that dispatch latency should be minimized. Further, without modification, their system does not handle processes with interactive *and* compute phases. A process that has slept for a long time and wakes up will dominate the CPU for an extended duration in virtue of holding exhaustible tickets. Finally, their algorithm for computing exhaustible tickets assumes that the total number of runnable tickets is constant. In reality, this number fluctuates as processes are created and destroyed, and sleep and wake up.

8 Future Work

Hybrid lottery scheduling heuristically identifies and rewards interactive processes by how much of their allocated CPU time they consume. However, some interactive processes, such as those that render graphics, also consume moderate amounts of CPU. Evans *et al.* suggest several methods based on past user action and window manager cooperation for an operating system to recognize interactive processes [6]. We wish to incorporate these methods into our scheduler so that once recognized, these processes can be allocated more tickets and preferentially scheduled.

In Section 3.1 we argued that kernel priorities are more desirable than ticket transfers for encouraging processes to release kernel resources quickly. However, to our knowledge, the chosen ordering of kernel priorities has not been rigorously studied and thus may not provide optimal performance in all cases. Ticket transfers are more dynamic because they enable additive and transitive transfers from multiple blocked processes. If the kernel can identify the loaner and borrower when a kernel resource is under contention, the kernel can perform this *implicit* ticket transfer transparently to the user. We wish to compare the throughput of different workloads with implicit ticket transfers versus kernel priorities.

9 Conclusion

This work incorporates into a lottery scheduler the specializations present in typical operating system schedulers to improve interactive response time and reduce kernel lock contention. We began with a straightforward implementation of lottery scheduling which enabled control over process execution rates and processor load insulation at the cost of interactive responsiveness relative to the FreeBSD scheduler baseline. To match the performance of the FreeBSD scheduler, we added kernel priorities, abbreviated quanta, and windowed ticket boost to lottery scheduling, resulting in a hybrid lottery scheduler. Further, user feedback prompted us to add support for the UNIX `nice` utility. These techniques have been applied without squandering the proportional-share resource management semantics. The principle technique used by these mechanisms is dynamic ticket adjustments that influence scheduling order while preserving CPU utilization targets.

Our measurements show that our optimized scheduler incurs more overhead than the FreeBSD scheduler, but that these differences are negligible even under heavy workloads. We achieve throughput and responsiveness nearly equal to the FreeBSD scheduler. Our system has been deployed to two production machines with success. This paper demonstrates that our hybrid lottery scheduler is a viable process scheduler for the workloads that we have tested.

Availability

Our hybrid lottery scheduler is available from http://www.-cs.cmu.edu/~dpetrou/hls.tgz. Included are two new kernel source files, a *context diff* that patches 14 existing kernel files, and the source for 10 user-level programs that interact with the scheduler.

Acknowledgments

We thank the anonymous reviewers for their careful and detailed comments. Thanks also go to UC Berkeley's Computer Science Undergraduate Association and FreeBSD Users' Group for permitting us to deploy our experimental kernel on their production machines. Our colleagues Remzi Arpaci-Dusseau, Joan Digney, Jason Flinn, Greg Ganger, Dushyanth Narayanan, and David Rochberg kindly reviewed drafts. Finally, thanks go to Aaron Smith for asking us to support `nice` semantics and David Greenman for providing us with `wcarchive` statistics.

References

[1] Andrea C. Arpaci-Dusseau and David E. Culler. Extending Proportional-Share Scheduling to a Network of Workstations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, June 1997.

[2] D. L. Black. Processors, priority, and policy: Mach scheduling for new environments. In *Proceedings of the USENIX 1991 Winter Conference*, pages 1–12, January 1991.

[3] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling*. Addison-Wesley, Reading, 1967.

[4] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. An experimental time-sharing system. In *Proceedings of the 1962 AFIPS Spring Joint Computer Conference*, volume 21, pages 335–344, May 1962.

[5] Raymond B. Essick. An event-based fair share scheduler. In *Proceedings of the Winter 1990 USENIX Conference*, pages 147–162. USENIX, January 1990.

[6] Steve Evans, Kevin Clarke, Dave Singleton, and Bart Smaalders. Optimizing Unix Resource Scheduling for User Interaction. In *Proceedings of the 1993 Summer USENIX*, pages 205–218. USENIX, June 1993.

[7] The FreeBSD Operating System, 1999. See http://www.-freebsd.org/.

[8] Berny Goodheart and James Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4, an Open Systems Design*. Prentice-Hall, 1994.

[9] Joseph L. Hellerstein. Achieving Service Rate Objectives with Decay Usage Scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, August 1993.

[10] Kevin Jeffay, F. Donelson Smith, Arun Moorthy, and James Anderson. Proportional share scheduling of operating system services for real-time applications. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, December 1998.

[11] L. Kleinrock. A continuum of time-sharing scheduling. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 453–458, 1970.

[12] J. Larmouth. Scheduling for immediate turnround. *Software—Practice and Experience*, 8(5):559–578, September/October 1978.

[13] J. M. McKinney. A survey of analytical time-sharing models. *ACM Computing Surveys*, 1, 2:105–116, 1969.

[14] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, Inc., 1996.

[15] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley Publishing Co., 2nd edition, 1992.

[16] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[17] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy Baruah, Johannes Gehrke, and C. Greg Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *IEEE Real-Time Systems Symposium*, December 1996.

[18] Jeffrey H. Straathof, Ashok K. Thareja, and Ashok K. Agrawala. UNIX scheduling for large systems. In *Proceedings of the USENIX 1986 Winter Conference*, pages 111–139. USENIX, Winter 1986.

[19] David G. Sullivan, Robert Haas, and Margo I. Seltzer. Tickets and currencies revisited: Extensions to multi-resource lottery scheduling. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, March 1999.

[20] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, September 1995.

[21] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, November 14–17 1994.

[22] Carl A. Waldspurger and William E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Mangement. Technical Report MIT/LCS/TM-528, Massachusetts Institute of Technology, MIT Laboratory for Computer Science, June 1995.

Retrofitting Quality of Service into a Time-Sharing Operating System

John Bruno, José Brustoloni, Eran Gabber, Banu Özden and Abraham Silberschatz
Information Sciences Research Center
Lucent Technologies — Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974, USA
{jbruno, jcb, eran, ozden, avi}@research.bell-labs.com

Abstract

Theoretical aspects of proportional share schedulers have received considerable attention recently. We contribute practical considerations on how to retrofit such schedulers into mainstream time-sharing systems. In particular, we propose `/reserv`, a uniform API for hierarchical proportional resource sharing. The central idea in `/reserv` is associating resource reservations with *references* to shared objects (and not with the objects themselves). We discuss in detail the implementation of `/reserv` and several proportional share schedulers on FreeBSD; the modified system is called *Eclipse/BSD*. Our experiments demonstrate that the proposed modifications allow selected applications to isolate their (or their clients') performance from CPU, disk, or network overloads caused by other applications. This capability is increasingly important for soft real-time, multimedia, Web, and distributed client-server applications.

1 Introduction

On a typical system, multiple applications may contend for the same physical resources, such as CPU, memory, and disk or network bandwidth. An important goal for an operating system is therefore to schedule requests from different applications so that each application and the system as a whole perform well.

Resource management schemes of time-sharing operating systems, such as Unix [15] and Windows NT [8], often achieve acceptably low response time and high system throughput for time-sharing workloads. However, as explained in the following paragraphs, several trends make those schemes increasingly inappropriate.

First, many workloads now include real-time applications (e.g., multimedia). Unlike time-sharing applications, real-time ones must have their requests processed within certain performance bounds (e.g., minimum throughput). To support real-time applications correctly under arbitrary system load, the operating system must perform *admission control* and offer *quality of service* (QoS) guarantees: The operating system admits a request only if the operating system has set aside enough resources to process the request within the specified performance bounds.

Second, even for purely time-sharing workloads, the trend toward distributed client-server architectures increases the importance of *fairness*, that is, of preventing certain clients from monopolizing system resources. The fairness of time-sharing systems can be quite spotty. For example, time-sharing systems typically cannot isolate the performance of a Web site from that of other Web sites hosted on the same system. If one of the sites becomes very popular, the performance of the other sites may become unacceptably (and unfairly) poor.

Finally, the same trend toward client-server architectures also makes it necessary to manage resources *hierarchically*, that is, recursively allowing each client to grant to its servers part of the client's resources. For example, new Web and other user-level servers often need mechanisms for processing client requests with specified QoS and/or fairness bounds. However, time-sharing operating systems usually do not provide such mechanisms.

The mentioned shortcomings of time-sharing operating systems have motivated considerable recent

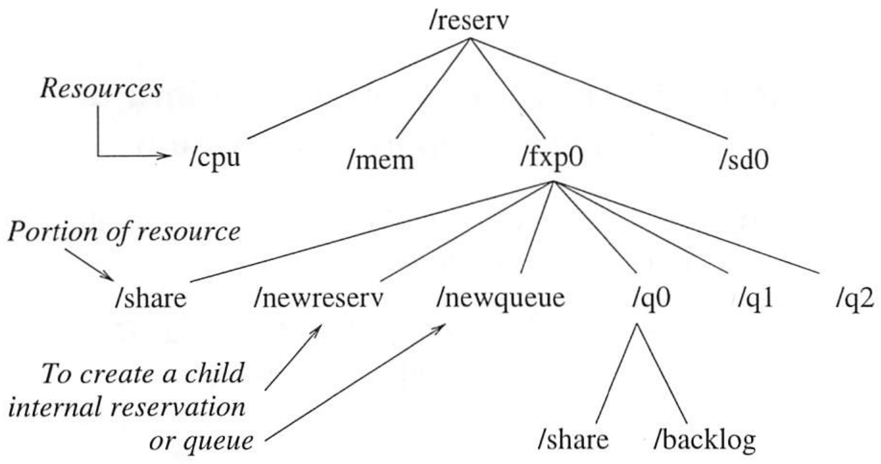


Figure 1: Eclipse/BSD's /reserv file system allows applications to create resource reservations.

work on new algorithms for CPU [14, 11, 13, 17, 6], disk [20, 21, 4], and network [2, 3, 12, 23] scheduling. In particular, we recently proposed MTR-LS, a new CPU scheduling algorithm with demonstrated throughput, delay, and fairness guarantees [6]. MTR-LS is an example of a *proportional share* scheduler. Proportional share schedulers stand out for their sound theoretical foundations [22, 11, 6, 2, 3, 23].

This paper considers the practical aspects of how to integrate proportional share schedulers into mainstream operating systems. We contribute a new application programming interface (API) for hierarchical proportional resource sharing: the `/reserv` file system. We discuss in detail the implementation of `/reserv` and several proportional share schedulers (MTR-LS, YFQ, H-WF^{2}Q) on FreeBSD. (FreeBSD is a freely available derivative of 4.4 BSD Unix; other Unix variants, Windows NT, and other time-sharing operating systems could be similarly modified.) We call the modified FreeBSD system *Eclipse/BSD*, as opposed to the *Eclipse/Plan 9* system used in our previous MTR-LS work [6] (where the distinction is obvious or unimportant, we will say simply *Eclipse*). Our experiments demonstrate how Eclipse/BSD's `/reserv` API and schedulers improve on FreeBSD's, providing QoS guarantees, fairness, and hierarchical resource management.

The rest of this paper is organized as follows. Section 2 describes the Eclipse/BSD resource management model and its retrofitting into FreeBSD. Section 3 discusses the scheduling algorithms used in Eclipse/BSD. Section 4 shows that Eclipse/BSD implementation requires only a modest amount of changes to FreeBSD. Experiments in Section 5 illustrate how Eclipse/BSD scheduling improves the isolation between Web sites hosted on the same system. Section 6 discusses related and future work, and Section 7 concludes.

2 Resource management model

This section describes the Eclipse/BSD hierarchical resource management model and its implementation on FreeBSD, including Eclipse/BSD's `/reserv` API.

2.1 Resource reservations

Eclipse/BSD applications obtain a desired quality of service by initially acquiring a *resource reservation* for each required physical resource. Physical resources include CPU, memory, disks, and network interfaces, each managed by a scheduler. A resource reservation specifies a fraction of the resource set aside for exclusive use by one or more processes. Applications can subdivide resource reservations hierarchically. Admission control guarantees that reservations do not exceed resources. Eclipse/BSD's schedulers share fractions of the respective resource fairly among all applications currently using the resource, as explained in the rest of this subsection.

Applications specify resource reservations as directories in a new file system mounted under `/reserv`. Each independently scheduled resource in the system corresponds to a directory under `/reserv`: `/reserv/cpu` (CPU), `/reserv/mem` (physical memory), `/reserv/fxp0` (network interface 0), `/reserv/sd0` (disk 0), and so on, as shown in Figure 1. Devices with multiple independently scheduled resources correspond to multiple directories, whereas multiple jointly scheduled resources (e.g., mirrored disks) correspond to a single directory.

A resource reservation r is called an *internal reservation* if it can have children, or a *queue* otherwise. r's parent p is always either `/reserv` or another reservation for the same resource. Each resource reservation r contains a `share` file that specifies two values: m_r, the minimum absolute value of the resources that r obtains from p, and ϕ_r, the weight with which r shares p's resources. m_r is specified in units appropriate to the respective resource (e.g., SPECint95 for CPU, bytes for physical memory, or Kbps for disk or network interfaces). If p is `/reserv`, $m_r = V$, the entirety of the resource, and ϕ_r is 100%. The amount of resources apportioned to a reservation r, v_r, depends dynamically on what reservations actually are being used. Every request arriving at a scheduler must specify a queue for processing that request; the request is said to *use* that queue. Schedulers enqueue and service in FIFO order requests that use the same queue. A reservation r is said to be *busy* while there is at least one request that uses r or a descendent of r.

If a resource reservation r is internal, then it also contains the files `newreserv` and `newqueue`. By opening either of these files, an application creates an internal reservation or queue that is r's child, respectively. The open call returns the file descriptor of the newly created `share` file, initialized with $m_r = 0$ and $\phi_r = 0$. Internal reservations thus created are consecutively numbered `r0`, `r1`, and so on, whereas queues are numbered `q0`, `q1`, and so on.

If resource reservation r is a queue, then it also contains the file `backlog`. Writing into `backlog` clears the number of requests served and amount of service provided and sets the maximum number of requests and amount of service that may concurrently be waiting in the queue. Reading from `backlog` returns the number of requests served and the amount of service provided (in units appropriate to the respective resource, e.g. CPU time or bytes).

Eclipse/BSD prevents reservations from exceeding resources as follows. Let S_p be the set of p's children and $M_{S_p} = \sum_{i \in S_p} m_i$. Then writing into the `share` file of $r \in S_p$ is subject to the following admission control rule: the call fails if p is `/reserv` (i.e., the entirety of the resource has a fixed value), $m_p < M_{S_p}$ (i.e., a parent's minimum resources must at least equal the sum of its children's minima after the attempted write), or $\phi_r < 0$ (i.e., weights must be non-negative).

Eclipse/BSD shares resources fairly according to the weights of the busy reservations. If reservation r is not busy, then its apportionment is $v_r = 0$. Otherwise, let p be the parent of r, B_p be the set of p's busy children, and $\Phi_{B_p} = \sum_{i \in B_p} \phi_i$. If p is `/reserv`, then:

$$v_r = V \tag{1}$$

where V is the entirety of the resource, else:

$$v_r = \frac{\phi_r}{\Phi_{B_p}} v_p \tag{2}$$

2.2 Reservation domains and root reservations

This subsection defines what resource reservations each process is allowed to create or use.

In Eclipse/BSD, a process P's *reservation domain* is the list of P's internal *root reservations*, one for for each resource[1]. Queue `q0` of process P's root reservation r is called P's *default queue* for the respective resource. A process P can list any directory under `/reserv` and open and read any `share` or `backlog` file, but can write on `share` or `backlog` files or open `newreserv` or `newqueue` files (i.e., create children) only in reservations that are equal to or descend from one of P's root reservations.

The reservation domain of a process `pid` is represented by a new read-only file, `/proc/pid/rdom`, added to FreeBSD's `proc` file system (where `rdom` stands for "reservation domain"). For example, `/proc/103/rdom` could contain:

```
/reserv/cpu/r2 /reserv/mem/r1
/reserv/fxp0/r0 /reserv/sd0/r3
```

meaning that process 103 has root CPU reservation r2, root memory reservation r1, root network reservation r0, and root disk reservation r3.

[1]Note that our current concept of reservation domain is somewhat different from that in our previous work [6].

If process 104 is in the same reservation domain, /proc/104/rdom would have the same contents. The reservation domain of the current process is also named /proc/curproc/rdom.

The reservation domain of processes spawned by a process pid is given by the new file /proc/pid/crdom (where crdom stands for "child reservation domain"). When a child is forked, its rdom and crdom files are initialized to the contents of the parent's crdom file. File /proc/pid/crdom is writable by any process with the same effective user id as that of process pid, or by the superuser. Writing into crdom files is checked for consistency and may fail: For each root reservation r in /proc/pid/rdom, /proc/pid/crdom must contain an internal reservation r' that is equal to or descends from r.

2.3 Request tagging

In Eclipse/BSD, every request arriving at a scheduler must be tagged with the queue used for that request, as explained in this section.

Resource reservations often cannot simply be associated with shared objects because different clients' requests may specify the same object but different queues. For example, two processes may be in different reservation domains and each need to use a different disk queue to access a shared file, or a different network output link queue to send packets over a shared socket. It would be difficult to compound reservations used on the same object correctly if reservations were associated with the object, since then one client could benefit from another client's reservations.

Therefore, Eclipse/BSD queues are associated with *references* to shared objects, rather than the shared objects themselves (e.g., process, memory object, vnode, or socket). This is accomplished by modifying FreeBSD data structures as follows:

- The CPU scheduler manages *activations* instead of processes. An activation points to a process *and* to the CPU queue in which that process should run.
- The *memory region* structure points to the region's memory object *and* memory queue.
- The *file descriptor* structure points to the file (and thereby to the vnode or socket) *and* to the device queue used for I/O on that file descriptor.

CPU, memory, and device queue pointers are always initialized to the process's default queue for the respective resource. Queue pointers can subsequently be modified only to descendents of the process's root reservation for the respective resource. Initialization and modification of queue pointers occur as follows:

- The initial activation created when a process P is spawned has CPU queue pointer according to the crdom file of P's parent. P can subsequently create children of its CPU root reservation, e.g. to process each client's requests. P can switch directly from one CPU queue to another by using a new system call, activation_switch. Alternatively, P can spawn new processes that run on CPU queues according to P's crdom file.
- The memory queue pointer of a region R is initialized when R is allocated, and can subsequently be modified using a new system call, mreserv, with region address, length, and name of the new memory queue as arguments.
- The device queue pointer of a file descriptor fd is initialized: for vnodes, at open time; for connected sockets, at connect or accept time; for unconnected sockets, at sendto or sendmsg time if fd's device queue pointer has not yet been initialized. A new command to the fcntl system call, F_QUEUE_GET, returns the name of the queue to which fd currently points. The queue pointer can subsequently be modified using the new command F_QUEUE_SET to the fcntl system call, with the name of the new device queue as argument.

Additionally, I/O request data structures (including uio for all I/O, mbuf for all network output, and buf for disk input that misses in the buffer cache and for all disk output) gain a pointer to the queue they use. Eclipse/BSD copies a file descriptor's queue pointer to the I/O requests generated using that file descriptor.

2.4 Reservation garbage collection

The previous subsections described how resource reservations are created and used; this subsection explains how they are destroyed.

Each resource reservation has a reference count equal to the number of times the reservation appears in an `rdom` or `crdom` file or is pointed by an activation, memory region, or file descriptor. A process's `rdom` and `crdom` files are created when the process is forked and are destroyed when the process exits. The file descriptor of a `share` file in the `/reserv` file system points to the respective resource reservation; additionally, as described in the previous subsection, file descriptors for vnodes and sockets also point to the resource reservations they use. Eclipse/BSD updates reservation reference counts on process `fork` and `exit`, `activation_switch`, memory region allocation and deallocation, `mreserv`, file `open` or `close`, socket `connect` or `accept`, `sendto`, `sendmsg`, and `fcntl` F_QUEUE_SET.

A `GC` flag determines whether a resource reservation should be garbage-collected when the number of references to the reservation drops to zero. When a resource reservation is created, its `GC` flag is enabled, but a privileged process can disable it. New commands to the `fcntl` system call, F_COLLECT_SET and F_COLLECT_GET, can be used on the file descriptor of a reservation's `share` file to set or get the reservation's `GC` flag.

Garbage collection of a queue q may need to be deferred. If q is being used by at least one request, q cannot be removed immediately; instead, q's REMOVE_WHEN_EMPTY flag is set. When the last request that uses q completes and q's REMOVE_WHEN_EMPTY flag is set, if q's reference count is still zero, the scheduler garbage-collects q, else the scheduler resets the flag.

3 Schedulers

The `/reserv` API described in the previous section provides an interface to proportional share schedulers. Eclipse/BSD incorporates a proportional share scheduler for each resource, as discussed in this section.

3.1 MTR-LS

Eclipse/BSD's CPU scheduler uses the MTR-LS (Move-To-Rear List Scheduling) algorithm [6]. When a process blocks (e.g., waiting for I/O), MTR-LS keeps the unused portion of the process's quota in the same position in the scheduling list, unlike the Weighted Round Robin (WRR) algorithm, which removes the process from the runnable list and, when the process becomes runnable again, places it back at the tail of the list. Consequently, MTR-LS may delay I/O-bound processes much less than does WRR. MTR-LS may also provide greater throughput than does WRR, whose scheduling delays may prevent I/O-bound processes from from fully utilizing their CPU reservations.

MTR-LS was specifically designed for CPU scheduling, where the time necessary to process a request cannot be predicted. To the best of our knowledge, MTR-LS is the only algorithm that provides the optimal cumulative service guarantee [6] when the durations of service requests are unknown *a priori*. However, MTR-LS assumes that requests can be preempted either at any instant or at fixed intervals. This is true of CPU scheduling, but usually is not true of disk or network scheduling, where requests cannot be preempted after they start and may take varying time to complete. Therefore, Eclipse/BSD uses other algorithms for I/O scheduling.

3.2 YFQ

Eclipse/BSD's I/O schedulers use approximations to the GPS (Generalized Processor Sharing) [18] model. GPS assumes an ideal "fluid" system where each backlogged "flow" in the system instantaneously receives service in proportion to the flow's share and inversely proportionally to the sum of the shares of all backlogged flows (where a backlogged flow is analogous to a busy queue). GPS cannot be precisely implemented for I/O because typically (1) I/O servers can only service one request at a time and (2) an I/O request cannot be preempted once service on it begins. GPS approximations estimate the time necessary for servicing each request and interleave requests from different queues so that each queue receives service proportionally to its share (although not instantaneously). However, the necessary time estimates may be difficult to compute precisely because GPS's rate of service for each flow

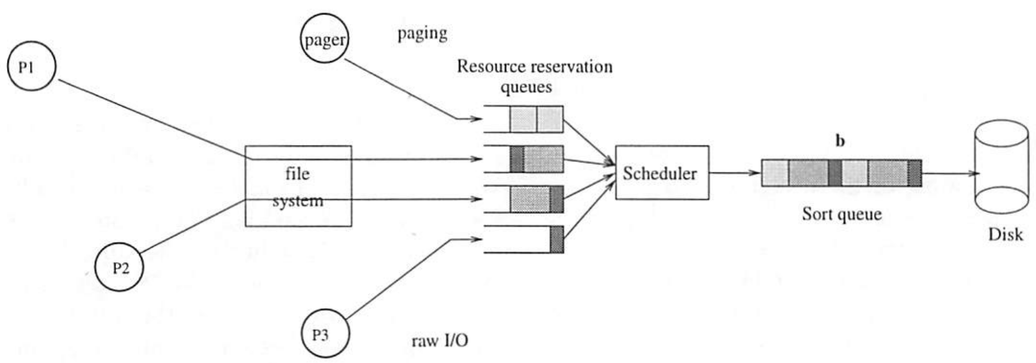


Figure 2: The sort queue allows the disk driver or disk to reorder requests and minimize disk latency and seek overheads.

depends on what flows are backlogged at each instant [3].

Eclipse/BSD's disk scheduler uses a new GPS approximation, the YFQ (Yet another Fair Queueing) algorithm [5], which can be implemented very efficiently. A resource is called *busy* if it has at least one busy queue, or *idle* otherwise. YFQ associates a *start tag*, S_i, and a *finish tag*, F_i, with each queue q_i. S_i and F_i are initially zero. YFQ defines a *virtual work* function, $v(t)$, such that: (1) $v(0) = 0$; (2) While the resource is busy, $v(t)$ is the minimum of the start tags of its busy queues at time t; and (3) When the resource becomes idle, $v(t)$ is set to the maximum of all finish tags of the resource.

When a new request r_i that uses queue q_i arrives: (1) If q_i was previously empty, YFQ makes $S_i = \max(v(t), F_i)$ followed by $F_i = S_i + \frac{l_i}{w_i}$, where l_i is the data length of request r_i; and (2) YFQ appends r_i to q_i.

YFQ selects for servicing the request r_i at the head of the busy queue q_i with the smallest finish tag F_i. r_i remains at the head of q_i while r_i is being serviced. When r_i completes, YFQ dequeues it; if queue q_i is still non-empty, YFQ makes $S_i = F_i$ followed by $F_i = S_i + \frac{l'_i}{w_i}$, where l'_i is the data length of the request r'_i now at the head of q_i.

Selecting one request at a time, as described above, allows YFQ to approximate GPS quite well, providing good cumulative service, delay, and fairness guarantees. However, such guarantees may come at the cost of excessive disk latency and seek overheads, harming aggregate disk throughput. Therefore, YFQ can be configured to select up to b requests (a *batch*) at a time and place them in a *sort queue*, as shown in Figure 2. The disk driver or the disk itself may reorder requests within a batch so as to minimize disk latency and seek overheads.

3.3 WF^{2}Q

Eclipse/BSD's network output link scheduler uses the hierarchical WF^{2}Q (Worst-case Fair Weighted Fair Queueing) algorithm [3]. This algorithm is similar to an earlier GPS approximation, WFQ (Weighted Fair Queueing) [9]. However, unlike WFQ, WF^{2}Q does not schedule a packet until it is *eligible*, i.e., its transmission would have started under GPS. Consequently, WF^{2}Q has optimal worst-case fair index bound, making it a good choice for a hierarchical scheduler [3].

Note that neither YFQ nor WF^{2}Q could be used for CPU scheduling, since they assume that the time necessary to process a request can be estimated and they never preempt a request.

3.4 SRP

Eclipse/BSD employs SRP (Signaled Receiver Processing) [7] for network input processing. SRP demultiplexes incoming packets *before* network and higher-level protocol processing. Unlike FreeBSD's single IP input queue and input protocol processing at software interrupt level, SRP uses an unprocessed input queue (UIQ) per socket and processes input protocols in the context of the respective applications. If a socket's queue is full, SRP drops new packets for that socket immediately, unlike FreeBSD, which wastefully processes packets that will need to be dropped. Because SRP processes protocols in the context of the respective receiving applications, SRP can avoid *receive livelock* [16], a network input overload condition that prevents any packets from being processed by an application.

When SRP enqueues a packet into a socket's UIQ, SRP signals SIGUIQ to the applications that own that socket. The default action for SIGUIQ is to perform input protocol processing (asynchronously to the applications). However, applications can synchronize such processing by catching SIGUIQ and deferring protocol processing until a later input call (e.g., `recv`). Synchronous protocol processing may improve cache locality. Unlike LRP (Lazy Receive Processing) [10], SRP does not use separate kernel threads for asynchronous protocol processing (kernel threads are not available in FreeBSD).

4 Implementation

This brief section shows that Eclipse implementation does not require too many changes to the underlying time-sharing system.

Our current Eclipse/BSD implementation adds approximately 6500 lines of code to FreeBSD version 2.2.8: 2400 lines for the `reserv` file system and modifications to the `proc` file system, and 4100 lines for the new schedulers and their integration into the kernel. The kernel size in the GENERIC configuration is 1601351 bytes for FreeBSD and 1639297 bytes for Eclipse/BSD (an increase of only 38 KB).

5 Experimental results

This section demonstrates experimentally that applications can use Eclipse/BSD's `/reserv` API and CPU, disk, and network schedulers so as to obtain minimum performance guarantees, regardless of other load on the system.

We ran experiments on the configuration shown in Figure 3, where HTTP clients on nodes A to E make requests to the HTTP server on node S. Nodes A to C are Pentium Pro PC's running FreeBSD. Nodes D and E are Sun workstations running Solaris. The operating system varies only in node S, being either

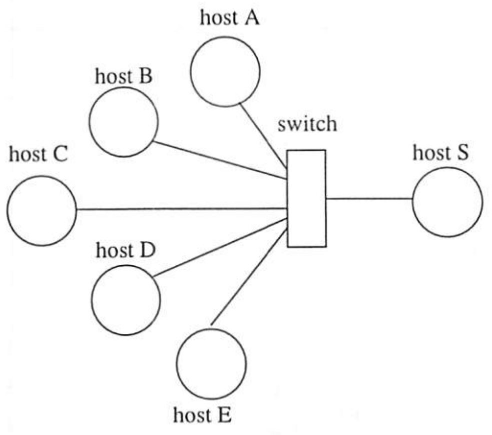


Figure 3: Node S is a Web server that hosts multiple sites on either FreeBSD or Eclipse/BSD.

FreeBSD or Eclipse/BSD. Node S is a PC with 266 MHz Pentium Pro CPU, 64 MB RAM, and 9 GB Seagate ST39173W fast wide SCSI disk. All nodes are connected by a Lucent P550 Cajun Ethernet switch (unless otherwise noted, at 10 Mbps). Node S runs the Apache 1.3.3 HTTP server and hosts multiple Web sites. Nodes A to E run client applications (some derived from the WebStone benchmark) that make requests to the server. At most ten clients run at each of the nodes A to E. Unless otherwise noted, all measurements are the averages of three runs.

Each experiment overloaded one of the server's resources, as described in the following subsections.

5.1 CPU scheduling

In the first experiment, an increasing number of clients continuously made CGI requests to either of two Web sites hosted at node S. Processing of each of these CGI requests consists of computing half a million random numbers (using `rand()`) and returning a 1 KB reply. Therefore, the bottleneck resource is the CPU. We measured the average throughput and response time (over three minutes) under the following scenarios: (1) The site of interest reserves 50% of the CPU and the competing site reserves 49% of the CPU; (2) The site of interest reserves 99% of the CPU; and (3) Both sites run in the same CPU reservation and reserve 99% of the CPU. Figure 4 shows the throughput of the site of interest when the latter has ten clients and the competing site has a varying number of clients, and Figure 5 shows the corresponding response times. Perfor-

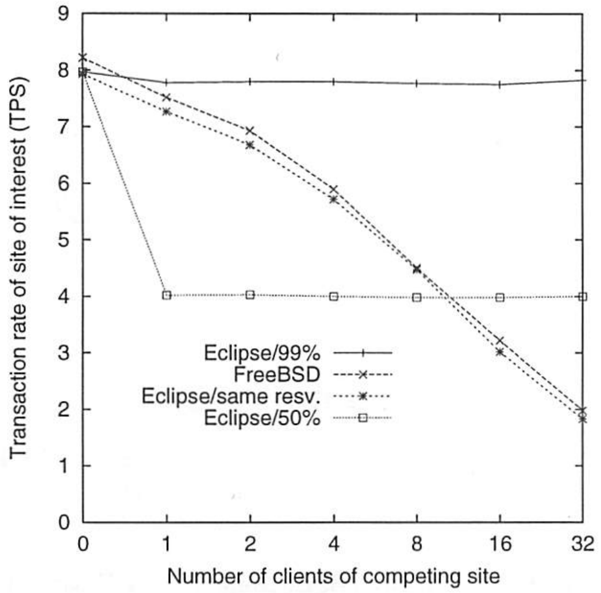


Figure 4: Appropriate CPU reservations can guarantee a minimum throughput for the site of interest.

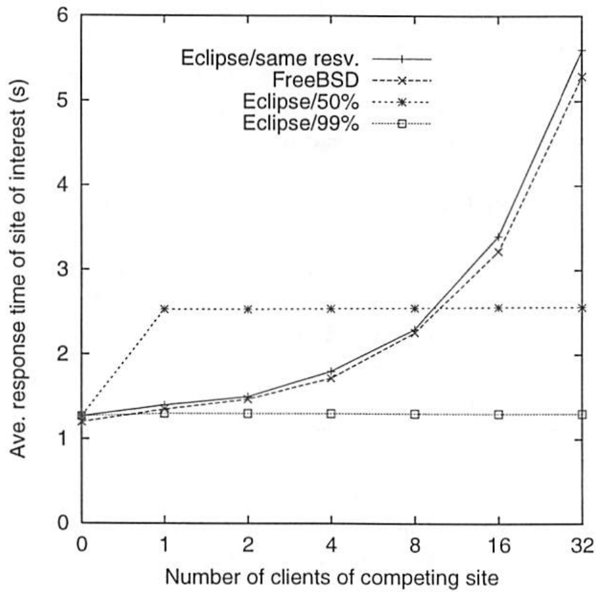


Figure 5: Appropriate CPU reservations can guarantee a maximum response time for the site of interest.

mance when both sites run in the same CPU reservation on Eclipse/BSD is roughly the same as performance on FreeBSD. When the site of interest reserves 99% of the CPU, its performance is essentially unaffected by other load. When the site of interest reserves 50% of the CPU, it still gets essentially all of the CPU if there is no other load, but, as would be expected, the throughput goes down by half and the response time doubles when there is other load. However, throughput and response time of the site of interest remain constant when further load is added, while on FreeBSD throughput decreases and response time increases without bound. This shows that FreeBSD and Eclipse/BSD are equally good if there is excess CPU capacity, but Eclipse/BSD can also guarantee a certain minimum CPU allocation (and consequently minimum throughput and maximum response time).

5.2 Disk scheduling

Again in the second experiment, an increasing number of clients continuously made CGI requests to either of two Web sites hosted at node S. However, these requests are I/O-intensive, consisting of reading a 100 MB file and returning a 10 KB reply. Because requests and replies are small and each request involves considerable disk I/O but little processing, the bottleneck resource is the disk. We reserved 50% of S's disk bandwidth to the Web site of interest and measured the latter's average throughput over three minutes. YFQ's sort queue was configured with a batch size of 4 requests. During the measurements, the site of interest had ten clients and the competing site had a varying number of clients. Figure 6 shows that in the absence of other load, Eclipse/BSD gives to the site of interest essentially all of the bottleneck resource, even though the site has only 50% reserved. When the load on the competing site increases, the throughput of the site of interest decreases. However, on Eclipse/BSD, the throughput bottoms out at roughly the reserved amount, whereas on FreeBSD the throughput decreases without bound. This shows that FreeBSD and Eclipse/BSD are equally good when there is excess disk bandwidth, but when bandwidth is scarce, Eclipse/BSD is also able to guarantee a minimum disk bandwidth allocation.

5.3 Output link scheduling

In the third experiment, an increasing number of clients continuously requested the same 1.5 MB document from either of two Web sites hosted at node S. Given that requests are much smaller than replies, little processing is required per request, and the re-

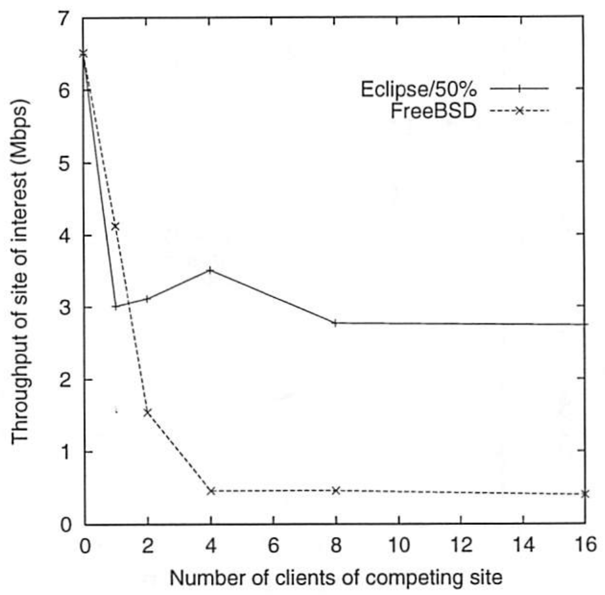


Figure 6: The site of interest gets at least its reserved fraction (50%) of the disk bandwidth.

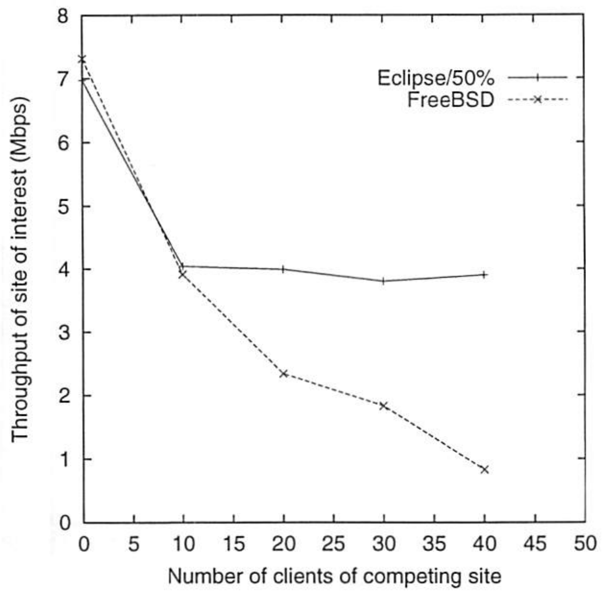


Figure 7: The site of interest gets at least its reserved fraction (50%) of the output link bandwidth.

quested document fits easily in the node S's buffer cache, the bottleneck resource is S's network output link. We reserved 50% of S's output link bandwidth to the Web site of interest and measured the latter's average throughput over three minutes. During the measurements, the site of interest had ten clients and the competing site had a varying number of clients. Figure 7 shows the results, which are very similar to those of Figure 6, where the disk is the bottleneck. FreeBSD and Eclipse/BSD are equally good when there is excess output link bandwidth, but when bandwidth is scarce, Eclipse/BSD is also able to guarantee a minimum output link bandwidth allocation.

5.4 Input link scheduling

The final set of experiments addresses network reception overload. In these experiments, the network operated at 100 Mbps full-duplex, and measurements are the averages of five runs.

In the fourth experiment, a client application sent 10-byte UDP packets at a fixed rate to a server application running at node S. Both on FreeBSD and on Eclipse/BSD, the server application received essentially all of the packets when the transmission rate was up to about 5600 packets per second (pkts/s). Above that transmission rate, as shown on Figure 8, the reception rate on Eclipse/BSD reached a plateau at around 5700 pkts/s. On FreeBSD, on the contrary, the reception rate dropped precipitously. This experiment shows that on Eclipse/BSD applications can make forward progress even when there is network reception overload, while on FreeBSD applications can enter livelock [16] in such situations. Eclipse/BSD prevents receive livelock because of SRP.

However, SRP cannot by itself guarantee that *important* applications will make forward progress. Eclipse/BSD can guarantee that by combining SRP and CPU reservations. In the fifth and final experiment, four different client applications sent 10-byte UDP packets at the same fixed rate to a different server application running on node S. We measured reception rates in two scenarios: (1) All four server applications reserved each 25% of the CPU; and (2) One server application reserved 97% of the CPU and the remaining server applications reserved 1% each. While the transmission rate was below 5600 pkts/s, essentially all packets were received. Reception rates increased slightly to 5900 pkts/s for a transmission rate of 28.5 Kpkts/s. Above that rate, results differ for the two scenarios, as shown in Figure 9. In the first scenario, reception rate goes down to about 1200 pkts/s. In the second scenario, the reception rate of the application with 97% of the CPU goes

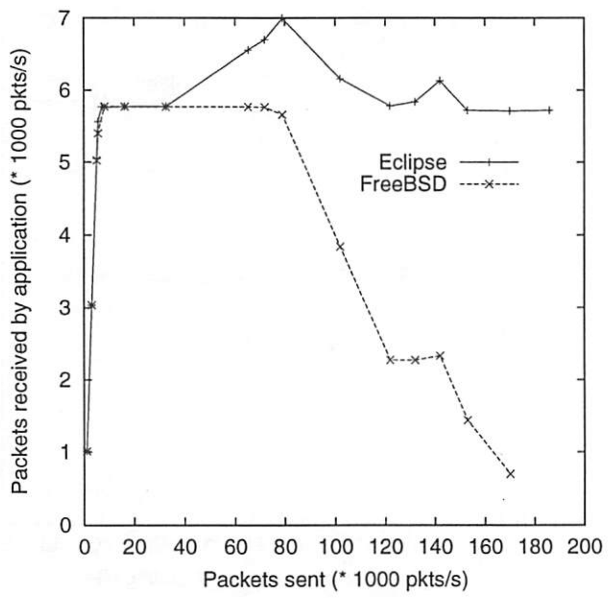


Figure 8: Eclipse/BSD avoids receive livelock.

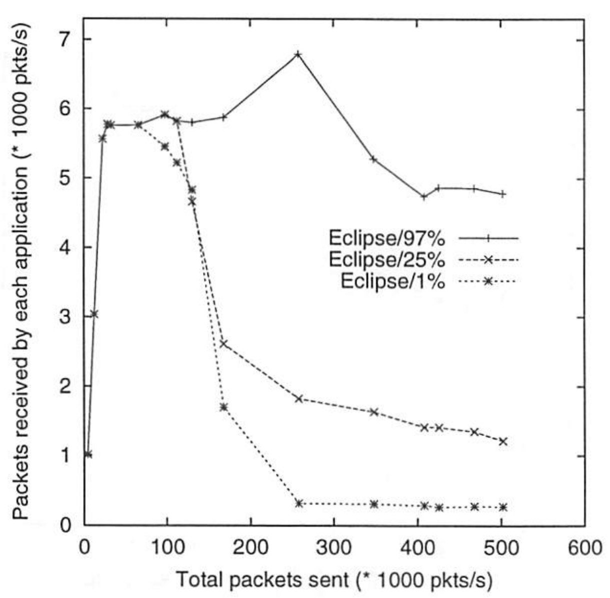


Figure 9: Eclipse/BSD guarantees forward progress according to CPU reservation.

down to about 4800 pkts/s, while the reception rate of the applications with 1% of the CPU goes down to about 160 pkts/s.

6 Related and future work

There are numerous recent works on proportional share scheduling [11, 6, 2, 3, 12, 23]. This paper complements those works by providing a uniform API for their schedulers and considering practical aspects of retrofitting them into mainstream operating systems. The API proposed here is also set apart by promoting uniformity not only across scheduling algorithms, but also across different resources.

The `/reserv` file system resembles many Plan 9 [19] APIs, which also use special file systems. We used Plan 9 in our previous MTR-LS work [6] but decided to replace it by FreeBSD because of FreeBSD's greater popularity and support for more current hardware.

Stride scheduling and the associated *currency* abstraction [24] can be used to group and isolate users, processes, or threads, much like the resource reservations discussed here. Another alternative is *resource containers* [1], which can isolate resources used by each client, whether within a single process or across multiple processes. Resource containers have been demonstrated primarily for priority-based CPU scheduling, not for hierarchical proportional sharing of different resources, as we advocate here. Our solutions for retrofitting reservations into a time-sharing system (e.g., how to associate reservations with references to shared objects, tag requests, and garbage collect reservations) may be useful also in conjunction with those frameworks.

Nemesis [14] is an operating system that uses a radical new architecture in order to eliminate QoS *crosstalk*, i.e., the degradation of one application's performance due to the load on another application. The Nemesis kernel provides only scheduling, and most other operating system services are implemented as libraries that are linked with applications and run in each application's address space. Eclipse/BSD attempts to provide similar isolation in a conventional monolithic architecture, requiring comparatively much less implementation effort.

SMART [17] is a hierarchical CPU scheduling algorithm that supports both hard real-time and conventional time-sharing applications, adjusts well to overload, and can notify applications when their deadlines cannot be met. Rialto [13] combines CPU reservations and time constraints into a scheduling graph that is used by a run-time scheduler to provide strong CPU guarantees. While SMART and Rialto target especially hard real-time CPU scheduling, the work presented here addresses mostly soft real-time scheduling of different resources and the integration of such scheduling into conventional systems.

7 Conclusions

We described how Eclipse/BSD applications can obtain resource reservations and thereby guarantee a desired quality of service for themselves or for their clients. Eclipse/BSD's API, `/reserv`, provides a simple, uniform interface to hierarchical proportional sharing of system resources. We discussed the different schedulers used in Eclipse/BSD and demonstrated experimentally that they can isolate the performance of selected applications from CPU, disk, or network overloads caused by other applications. Eclipse/BSD was implemented by making straightforward modifications to FreeBSD and greatly improves the system's ability to provide QoS guarantees, fairness, and hierarchical resource management. We believe that other common time-sharing systems would benefit from similar modifications.

Acknowledgments

We thank Mary McShea, Amit Singh, and Josep Blanquer for their help in the implementation and experiments, and the anonymous referees for their valuable comments.

References

[1] G. Banga, P. Druschel and J. Mogul. "Resource Containers: A New Facility for Resource Management in Server Systems", in *Proc. OSDI'99*, USENIX, Feb. 1999.

[2] J. Bennet and H. Zhang. "WF^{2}Q: Worst-Case Fair Weighted Fair Queueing", in *Proc. INFOCOM'96*, IEEE, Mar. 1996, pp. 120-128.

[3] J. Bennet and H. Zhang. "Hierarchical Packet Fair Queueing Algorithms", in *Proc. SIGCOMM'96*, ACM, Aug. 1996.

[4] P. Barham. "A Fresh Approach to File System Quality of Service", in *Proc. NOSSDAV'97*, IEEE, May 1997, pp. 119-128.

[5] J. Bruno, J. Brustoloni, E. Gabber, B. Özden and A. Silberschatz. "Disk Scheduling with Quality of Service Guarantees", to appear in *Proc. ICMCS'99*, IEEE, June 1999.

[6] J. Bruno, E. Gabber, B. Özden and A. Silberschatz. "The Eclipse Operating System: Providing Quality of Service via Reservation Domains", in *Proc. Annual Tech. Conf.*, USENIX, June 1998, pp. 235-246.

[7] J. Brustoloni, E. Gabber and A. Silberschatz. "Signaled Receiver Processing", submitted for publication.

[8] H. Custer. "Inside Windows NT", Microsoft Press, 1993.

[9] A. Demers, S. Keshav and S. Shenker. "Design and Analysis of a Fair Queueing Algorithm", in *Proc. SIGCOMM'89*, ACM, Sept. 1989, pp. 1-12.

[10] P. Druschel and G. Banga. "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems", in *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 261-275.

[11] P. Goyal, X. Guo and H. Vin. "A Hierarchical CPU Scheduler for Multimedia Operating Systems", in *Proc. OSDI'96*, USENIX, Oct. 1996, pp. 107-121.

[12] P. Goyal, H. Vin and H. Chen. "Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks", in *Proc. SIGCOMM'96*, ACM, Aug. 1996.

[13] M. Jones, D. Ro su and M. Ro su. "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities", in *Proc. SOSP'97*, ACM, Oct. 1997, pp. 198-211.

[14] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns and E. Hyden. "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications", in *JSAC*, 14(7), IEEE, Sept. 1996, pp. 1280-1297.

[15] M. McKusick, K. Bostic, M. Karels and J. Quarterman. "The Design and Implementation of the 4.4 BSD Operating System", Addison-Wesley Pub. Co., Reading, MA, 1996.

[16] J. Mogul and K. K. Ramakrishnan. "Eliminating Receive Livelock in an Interrupt-Driven Kernel", in *Proc. Annual Tech. Conf.*, USENIX, 1996, pp. 99-111.

[17] J. Nieh and M. Lam. "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications", in *Proc. SOSP'97*, ACM, Oct. 1997, pp. 184-197.

[18] A. Parekh and R. Gallager. "A Generalized Processor Sharing Approach to Flow Control — The Single Node Case", in *Trans. Networking*, ACM/IEEE, 1(3):344-357, June 1993.

[19] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey and P. Winterbottom. "Plan 9 from Bell Labs", in *Computing Systems*, USENIX, 8(3):221-254, Summer 1995.

[20] P. Shenoy and H. Vin. "Cello: A Disk Scheduling Framework for Next Generation Operating Systems", in *Proc. SIGMETRICS'98*, ACM, June 1998.

[21] P. Shenoy. P. Goyal, S. Rao and H. Vin. "Design and Implementation of Symphony: An Integrated Multimedia File System", in *Proc. Multimedia Computing and Networking*, SPIE, Jan. 1998.

[22] D. Stiliadis and A. Varma. "Frame-Based Fair Queueing: A New Traffic Scheduling Algorithm for Packet-Switched Networks", Tech. Rep. UCSC-CRL-95-39, Univ. Calif. Santa Cruz, July 1995.

[23] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke and C. G. Plaxton. "A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems", in *Proc. Real Time Systems Symp.*, IEEE, Dec. 1996.

[24] C. Waldspurger and W. Weihl. "An Object-Oriented Framework for Modular Resource Management', in *Proc. IWOOOS '96*, IEEE, Oct. 1996, pp. 138-143.

Adaptive Modem Connection Lifetimes

Fred Douglis* Tom Killian†
AT&T Labs–Research

Abstract

Internet Service Providers sometimes go to great lengths to minimize dial-up connection times, in order to make the best use of limited resources. Typically they disconnect users after a fixed period of complete inactivity, such as 10–15 minutes. We propose adaptive time-out policies that take past history into account, and we evaluate some of these policies using a trace from a production environment. We find that adaptive policies can reduce cumulative connection times and average simultaneous usage by about 10–20% compared to a conservative fixed threshold, in exchange for a moderate increase in the number of disconnections that inconvenience the user.

1 Introduction

In computers, as in real life, using resources often comes at a cost in proportion to the duration of use. That cost can typically be reduced by discontinuing use temporarily—assuming that the resource will not be needed for a period of time—and then using the resource again in exchange for some possible start-up overhead. Often, the decision to discontinue use is based on a fixed time-out interval: one waits for the resource to be idle long enough that one can assume it will continue to be idle a while longer, long enough to avoid antagonizing the user. It must also be idle long enough to amortize any start-up overhead and result in a net gain from discontinuing use.

One category of resource use with variable timeouts is anything that consumes energy on a battery-operated device such as a mobile computer. Examples include spinning down the disk [2, 3, 8], putting the CPU in a low-power or reduced-power state [10], and suspending the display. In the communications domain, an example of this tradeoff has been demonstrated in the IP-over-ATM area. One can use a variety of algorithms to decide when to relinquish a virtual circuit that is being used to transmit a sequence of IP datagrams [6]. An algorithmic approach that is common to many of these systems has been termed a "random walk," in which a parameter varies over time in response to past history [5].

Modems are another type of limited resource, with some interesting properties that make straightforward approaches somewhat problematic. With most telephone-modem-based Internet Service Providers (ISPs), a computer connects to the Internet via a pair of modems, one owned by the user and one owned by the ISP. The ISP provides a temporary IP address using the Point-to-Point Protocol (PPP) [9] or some other protocol. If the modem connection is terminated, the computer is not guaranteed to get the same IP address the next time it connects. This means that proactively disconnecting the modem will likely terminate any existing TCP connections using the older IP address.

Furthermore, even if the IP address is fixed, there is a significant delay in reestablishing a PPP connection: we have measured 30–40 seconds for dialing, training, and PPP negotiation. This delay can be annoying to the user, and it can also cause application-level or system-level timeouts to occur. This might result in a transient error (for instance, downloading a Web page) or a more serious one (such as terminating a telnet session, forcing it to be reestablished and losing any state in it).

At the same time, constant use of an ISP's modem is generally discouraged. One way to discourage use is an economic disincentive: about a year ago, AT&T, MCI, and other ISPs changed their "unlimited" Internet Access to have a limit of 150 hours per month before surcharges are applied. This limit was imposed because a small fraction of the user community would use the system virtually non-stop, forcing the ISPs to continually increase capacity or risk having other customers encounter busy signals.

Another way to limit modem use is to proactively disconnect "idle" users and suffer the consequences. It appears that many ISPs will disconnect a completely idle user after some fixed interval that varies in the range of about 10–60 minutes. But users do not like to suffer the 30–40 seconds of delay reconnecting to the ISP after being reconnected, nor the possibility of a busy signal, nor

*Email: `douglis@research.att.com`
†Email: `tom@research.att.com`

the possible loss of TCP sessions that are open but inactive and which get reset after a hangup. A simple solution, from their perspective, is to run a daemon that uses the modem periodically, more often than the usual timeout. Checking electronic mail is an obvious example of such a daemon. Since completely periodic use, such as checking mail every 5 minutes, could be detected and compensated for, a more sophisticated approach would be to access the modem at somewhat random intervals for as long as the client wishes to keep the modem connection alive. As a result, some ISPs drop connections after an extended period of activity, such as a day, regardless of ongoing use.

Our goal was to see whether another approach to disconnecting idle modems might reduce modem usage without significantly inconveniencing users. Here we apply an "adaptive" timeout technique, which was previously applied to the domain of disks [2], to the domain of modems. The basic approach is the same: an ISP would start with some timeout interval, and then vary that timeout interval for each client over time based on usage patterns. A shorter interval will generally reduce modem usage but be more susceptible to making a mistake, i.e., disconnecting a client at a time when it will be active again too soon to make disconnecting it worthwhile. One decreases the timeout when the previous interval successfully predicted a long enough idle interval, and increases it when the idle interval proved too short. One can also simultaneously track *patterns* of bursts of activity, for instance accesses for just a few seconds every 15 minutes, in order to predict the next idle interval and disconnect immediately. We will elaborate on these approaches, and quantify the outcomes, later in this paper.

The environment in which we apply adaptive modem timeouts is particularly conducive to proactive hangups, despite the problems mentioned above. First, it uses static IP addresses, so a modem can be disconnected and later reconnected without affecting running applications if the applications do not attempt communication during the downtime. (If they do communicate, they must have a long enough timeout to tolerate the reconnect delay.) Second, the downstream connection is unaffected by a disconnection. Packets from the ISP can reach the client, while acknowledgments or other traffic from the client to the ISP will be delayed during the reconnection. For small, transient communication from the ISP to the client, the client might not observe a communication delay. We discuss applicability to more traditional ISPs below in Section 4.5.

Trace-driven simulations indicate that varying the timeout adaptively has significant benefits over relatively short fixed timeout intervals (2–10 minutes). For instance, all the adaptive algorithms we studied resulted in many fewer bad choices about when to hang up the modem than a short fixed threshold, with at most the same cumulative connection time. The longest fixed threshold we considered, 15 minutes, substantially drops the number of bad choices by comparison to the adaptive and shorter fixed thresholds, in exchange for more connect time.

The rest of this paper is organized as follows. The next section discusses the metrics we use to evaluate the costs and benefits of modem disconnection. Section 3 discusses our target environment and the trace we collected from it. Section 4 discusses several different approaches to varying the timeout threshold. In Section 5, we describe the experiments we performed, the results of which appear in Section 6. Section 7 concludes.

2 Metrics

In deciding when to disconnect a modem, one must balance the inconvenience to the user against the benefit to the ISP due to reclaiming the modem.

2.1 Inconvenience

From the user's perspective, each time the modem disconnects there is potentially some inconvenience. Waiting several seconds (typically on the order of a half-minute) for reconnection is a mild annoyance if the user has not used the network for a long time, but it would be a greater problem if:

- the idle time was very brief, or
- the reconnection adversely affected running processes (such as terminating a telnet session).

Occasional annoyances are probably fine, while frequent ones would be intolerable and should be avoided.

Our passive monitoring of the modem pool does not tell us when a connection is terminated, so we focus on idle time. In our initial experiments, we use a parameterized idle-time threshold. If the modem has been idle that long after being disconnected, then disconnecting was desirable. If it has not, then disconnecting was undesirable. In the adaptive disk spin-down work [2] on which we model our system, undesirable disk spin-downs were referred to as "bumps," and we adopt that terminology here. We use a default of 5 minutes of inactivity after a disconnect as the required idle time to avoid a bump, and then compare this with a more conservative 10-minute threshold.

In [2], a bump was considered a binary event: either a bad spin-up occurred, or it did not. The study alluded to considering some bumps as more egregious than others. We apply that reasoning here by counting bumps either as binary events or as fractional numbers. In the former case, we charge the same amount (1) any time the idle-time threshold is not met. In the latter case, we charge $1 - \frac{I}{T}$, where I is the idle time since disconnect and T is the threshold. Thus a reconnect immediately after a disconnect is charged 1 unit, whereas a reconnect just before the threshold T is crossed is charged almost nothing. One can view these two values as a count of the total *number* of bumps and the *severity* of the bumps.

2.2 Benefits

Benefits to the ISP accrue when a modem (and the phone line to which it is attached—there is effectively a one-to-one ratio) is used less. In the event that an ISP is charged in proportion to connect time, the total *connect time* across all users is relevant. (Examples of this in the general ISP market are rare, but some services that function effectively as ISPs do observe this property. For instance, AT&T has a corporate network for employees with a toll-free number, and it pays per-minute charges which are in turn charged back to the users of the dial-up service.)

Another benefit is the potential ability to reduce the total number of modems in the ISP (or conversely, to serve more users with the same number of modems). We consider both the *average simultaneous user count* and the *maximum simultaneous user count*. The average is an indication of how one could provision the modem pool to satisfy demand most of the time while rarely running out of resources. The maximum shows how to provision the modem pool in order never to run out of resources at all. Another useful metric might be the 90th or 95th percentiles, rather than the mean, but we have not yet considered percentiles other than 100%.

3 Environment and Trace Collection

This study was performed in the context of the Speedy Asymmetric Intranet Link (SAIL) project in AT&T Labs. SAIL uses cable modems to transmit data at high-speed to home users, with the "upstream" link over a 28.8 kbps telephone modem. (This configuration was for some time typical of cable modems, with only about 20% of cable plants supporting two-way communication as of June 1998 [7]. Resource allocation in two-way cable is a similar problem, however, and may benefit from our approach.) The upstream link uses PPP [9], with dynamically assigned IP addresses. IP endpoints, however, deal only with the downstream cable-modem addresses, and these are statically assigned. Using static downstream addresses enables the modem pool to disconnect a user after a relatively short period of inactivity, usually 10–15 minutes, without normally impacting connections beyond the dial-up overhead. It takes 30–40 seconds to reconnect over the telephone modem; the cable modem is virtually always accessible.

Figure 1 depicts the architecture of SAIL. SAIL connects AT&T Labs–Research with homes throughout the northern part of New Jersey, by distributing data over several cable head-ends. The upstream connections come in through modem pools at or near the cable head-ends and are backhauled into the AT&T Labs network. Downstream data flows over a collection of T1 lines and optical fiber to four regional cable television headends.

We collected a trace over a one-week period in May, 1998 by periodically polling each of four modem pools (a total of 168 modems) for activity. The modems would report which users were active, and how many bytes had been transferred since the most recent connection. Activity could be inferred by a change in the byte-count, while disconnection could be detected by the complete absence of a particular userid in the report. We encountered a total of 87 distinct users during the trace interval, with a maximum of 50 active users at any time. The raw ASCII trace (complete with some debugging information) was 246MB, which was preprocessed into a 3MB file consisting just of the active clients at each polling interval across all modem pools.

The polling granularity was set at 30 seconds, a somewhat arbitrary choice that was based on the balance between the desire not to load the modems unnecessarily and the desire for current statistics. With a 30-second granularity, one could not tell whether a newly detected connection was established just after the previous poll (i.e., 30 seconds previously), just before the current poll, or anytime in-between. We take the conservative approach of "charging" a connection from the earliest point when it may have become active, which we approximate as 29 seconds before the current polling interval.

Although the script that gathered this trace was capable of actually disconnecting modems, it acted primarily in a non-intrusive capacity. The reason for this was two-fold. First, acting on live connections permits one to apply only one policy, since after disconnecting a modem, one cannot wait a minute or two and disconnect it again. In practice, the modems use a fixed-threshold policy with a 15-minute timeout in the vast majority of cases and an infinite timeout in a few cases where users previously complained about unwanted disconnections. This would limit us to implementing policies that would disconnect

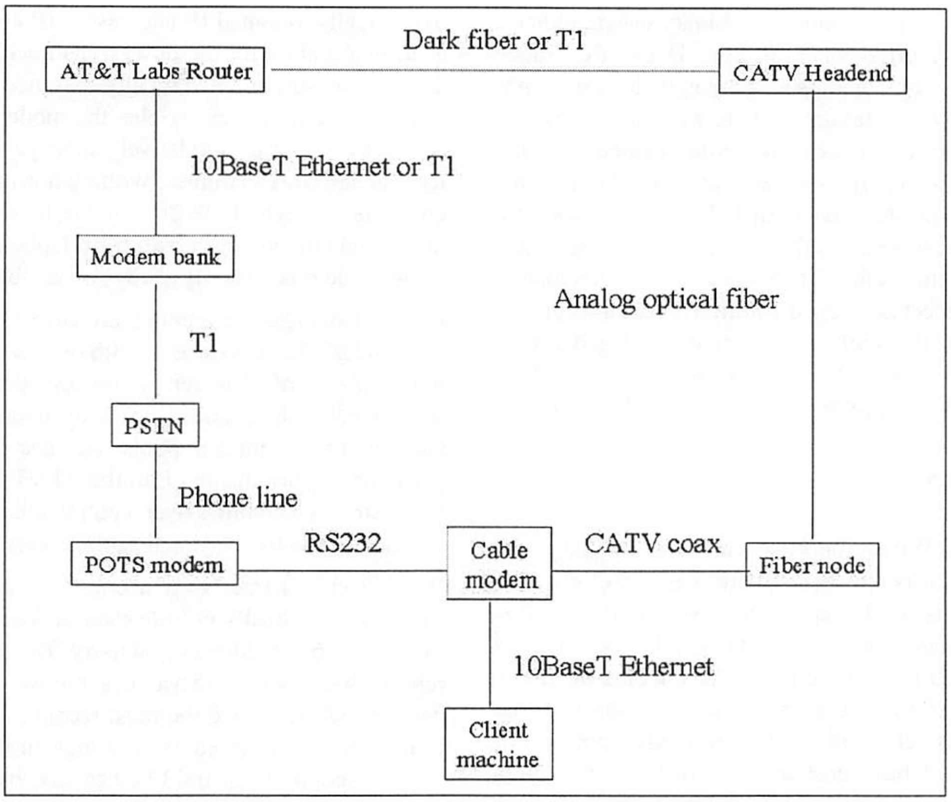


Figure 1: SAIL architecture.

in at most 15 minutes, since anything longer would have the real system "beat us to the punch" in disconnecting the user.

The second reason was a fear that disconnecting live users might upset them if we were too aggressive. We wanted to simulate the effect first. The sole exception to this "hands off" approach was the modem connection of one of the authors, which was disconnected using one of the simple adaptive schemes described below. Over the one-week collection interval, his modem was disconnected by the script just over 200 times, and 43 of them (21%) were deemed by the script to be premature: the subsequent idle time was not long enough. However, completely subjectively, at no time was the idle time since the previous access so short as to prove particularly annoying.

Finally, we grant that this trace has its limitations. It is a static snapshot of what people did, and not what they would do if the policies suggested here were in place. It traces users of a system with ample capacity (users would virtually never encounter a busy modem pool, or they might be more inclined to stay connected full-time). Users had no per-minute charges that they would pay out of pocket, though in some cases their company would pay toll charges on their behalf. The applications used by these individuals, and network access patterns, were potentially different from what "home users" of a commercial ISP would encounter—especially because the downstream bandwidth was much higher than two-way telephone modems would experience.

4 Variable Timeouts

In contrast to the fixed timeout intervals currently in use, we consider two types of variable timeout intervals, which are complementary. Adaptive timeouts use an interval that fluctuates over time as a function of past history. Predictive timeouts use a small window of past history to predict when the next access will follow the same pattern. In addition, we consider an off-line optimal timeout algorithm as a baseline against which to compare other approaches.

4.1 Adaptive Timeouts

Our adaptive algorithm attempts to keep the number of undesirable disconnects low, relative to total connect time. At any given time, each modem has a timeout T_m associated with it. If the modem has been idle for T_m seconds, it is disconnected. The next time the modem is used, the idle time since the disconnect, I, is compared against a minimum idle time M. If $I \geq M$, then the disconnect was "acceptable" and the threshold T_m is reduced. Otherwise, it was a "bump," and T_m is increased.

A key question with this approach is how much to adjust the threshold. We use the same approach as with adaptive disk spin-down [2], permitting both additive and multiplicative modifiers along with minimum and maximum values. A given policy specifies a starting threshold T_s, how to adjust on a acceptable disconnect or a bump, and a range. If the adjustments are additive, then we add a fixed amount on a bump, or subtract on an acceptable disconnect. We repeat the terminology of [2], using α_a or α_m to adjust on a bump (the subscript a denotes an additive adjustment and the subscript m denotes a multiplicative adjustment), while β_a and β_m apply in the case of an acceptable disconnect. As with disk spin-down, experience suggests that one should decrease the threshold slowly on success and increase it more rapidly on failure.

Multiplicative adjustments tend to affect the threshold more rapidly. For instance, we may multiply by 1.4 on a bad disconnect and divide by 1.1 on an acceptable one. The threshold will nearly double on back-to-back bumps.

The range is used as a sanity check. By preventing the threshold from falling below some minimum (e.g., 1 minute), we avoid entering a state in which a disconnect might occur through a perfectly normal gap between packets. By keeping it from rising above some maximum (e.g., 15 minutes), we prevent an adaptive algorithm from becoming strictly worse than the most conservative fixed threshold we would apply. (In addition, since the original trace applied a maximum, the trace would indicate a disconnect even if the replay would have kept the connection intact.)

4.2 Predictive Timeouts

In studying access patterns, it became apparent that some modems were used at regular intervals. For example, a host might check mail or perform some other sort of "keep-alive" at 15-minute intervals, and with a 15-minute fixed timeout, the modem might either never disconnect or repeatedly disconnect just moments before the next access. It seemed desirable to detect these patterns and disconnect quickly after each burst of activity, under the assumption that the next activity would not occur for many minutes. One must use a history buffer to look at some number of past accesses for a pattern; in our case we looked at the past 5 active and idle periods.

Being too aggressive in detecting these patterns could of course be a mistake. If one were to assume that because every recent interval of activity was brief (a few seconds), the next interval will be equally brief, then normal activity might also trigger a disconnect during a momentary lull. This situation is analogous to the case of a very short adaptive threshold, and is addressed the same way: a separate minimum idle time is established. In our simulations we used a 2-minute threshold, meaning that when periodic activity was detected, the timeout threshold would be dropped temporarily to 2 minutes if it were not already below that point.

What if the prediction is wrong? A wrong prediction would mean that past recent history was not a good predictor of the next access. We increment a counter, and stop predicting for a client if the count of incorrect predictions exceeds a threshold (currently 2 mistaken predictions). In that event, the adaptive modifiers to the threshold still apply, but we do not shortcut the threshold just because a pattern seems to develop.

4.3 Off-line Optimal

Given a trace, it is possible to look ahead to the next access and disconnect the modem if and only if the next access will be M seconds in the future. Disconnections with future knowledge are the best a system can do, and we refer to such disconnections as the "off-line optimal" policy since it can be done only with knowledge of future activity. (In fact, assuming the time cost of reconnecting is C, one could take this a step further and reestablish the connection C seconds before the next access [2], but we do not consider that here.) We simulated the off-line optimal threshold as a way to compare different algorithms and evaluate any additional room for improvement.

In the case of connect times, we use the total connect time for the off-line optimal as a baseline, and report other connect times as a ratio by comparison to the optimal. (In general, a 2-minute fixed threshold has less cumulative connect time than the optimal, since the optimal will not disconnect unless there is at least a 5-minute period of inactivity. However, it experiences an exceptionally high number of bumps.) The same holds true of the average and maximum number of modems in use over time, which are also normalized to the optimal.

The number of bumps is zero for the off-line optimal, so a ratio does not apply. We use absolute counts, displayed on a log scale, with the zero value for the optimal case not displayed in the graph but mentioned in each corresponding figure caption.

4.4 Required State

With fixed thresholds, the per-modem state at the ISP is minimal. The server pool tracks the last time when there was activity, and compares the idle time to some threshold. This threshold can either be the same for every user, or be configured separately for each user. (The latter is done by SAIL, permitting users who pay toll charges to disconnect more aggressively.) A separate threshold per user requires two values to be stored for each active modem rather than just one, but either way the state required is trivial.

Using an adaptive threshold, one needs the per-modem timeout, as well as a global or per-modem set of parameters for adjusting the timeout. One also must note not only the last time of activity but also the time when a modem disconnected, in order to determine whether a bump has occurred. Predictive timeouts, which use a buffer of historical information, require several idle time and active intervals for each modem. In no case is the per-modem state more than a few tens or hundreds of bytes per modem, however.

4.5 Applicability to General ISPs

In the Introduction, we described how SAIL's static addresses and asymmetric model was especially suitable for adaptive modem timeouts. In practice, ISPs usually use PPP [9] to assign IP addresses as users connect to the network. Without modifications to PPP, an ISP that proactively disconnected users after brief periods of inactivity could cause them to obtain a new IP address on each reconnection, which would cause existing connections to be terminated.

To address this issue, ISPs could either assign static IP addresses, or use a simple caching mechanism to reuse the last address. Static addresses would be unacceptable to a large ISP with many more inactive clients than active ones, but reserving an address for a period of time after a disconnection would be a simple extension. In fact, the Dynamic Host Configuration Protocol (DHCP) [4] uses *leases* to reuse the same address for the same client for a fixed period of time, and PPP could use a similar extension.

The asymmetric model does not have an analogy for traditional ISPs with all communication via the phone line, but the added benefits of instant downstream communication via the cable are relevant only in a limited set of scenarios and do not dramatically affect the benefits of adaptive timeouts.

5 Experiments

The fixed-threshold policies used disconnect thresholds of 2, 5, 10, and 15 minutes. When reported individually, these are designated by *fixed*N, for a value of N. In our graphs, the fixed policies are generally connected by a dashed line to highlight them by comparison to the adaptive policies, and because one can usually interpolate between two fixed thresholds. The adaptive policies used additive or multiplicative modifiers within ranges that approximated the fixed-threshold policies. When clustered, these are grouped by the type of modified and the range within which they vary. When shown separately, they are designated by a string of the form $\mathbf{S}\text{-}\beta_a\text{+}\alpha_a\mathbf{m}min\mathbf{M}max$ for additive modifiers, or $\mathbf{S}/\beta_m{*}\alpha_m\mathbf{m}min\mathbf{M}max$ for multiplicative ones. S is a starting threshold in minutes; all experiments reported here started with a 5-minute threshold.

The values for α and β appear in Table 1(a), and the ranges among which they varied appear in Table 1(b). Note that the values for β_m in the table refer to an amount by which to divide the current threshold. Note also that the ranges we considered varied more than the range of fixed thresholds we considered (2–15 minutes), because allowing *individual* modems to vary within the wider range can be beneficial even though fixing *all* modems to time out after 30 minutes would be counterproductive.

In addition to varying the modifiers and ranges, we varied the definition of a bump and the inclusion or exclusion of "workaholics," as described next.

5.1 Workaholics

Some clients are essentially always active. This may be because they are actively using the network, getting useful work done. In other cases the communication is "busy work" that is specifically intended to keep the connection alive. Either way, no method that is intended to modify the disconnection threshold is going to affect these clients. The more "workaholics" there are, the less overall benefit might accrue from modifying the behavior of the non-workaholics. (Barbará and Imieliński [1] refer to the latter as "sleepers.")

We identify workaholics by noting those modems that accumulated a total connection time of at least 80% of the entire trace, using the off-line optimal disconnection threshold. The simulator reports statistics across all modems, and also excluding those modems that we previously identified as workaholics. The number of workaholics is partially dependent on the definition of a bump; in our trace, with a 5-minute bump threshold, 15 hosts

Additive		Multiplicative	
α_a	β_a	α_m	β_m
5.00	−1.00	1.2	1.1
5.00	−2.00	1.4	1.1
3.00	−2.00	1.4	1.2

(a) Adjustment values. The left two columns list the additive values studied in this paper, while the right two list the multiplicative values. Additive times are in minutes.

Starting value	Minimum value $T_{\min}$	Maximum value $T_{\max}$
5	1	15
5	5	15
5	5	30

(b) Ranges of the adaptive disconnect threshold studied in this paper, in minutes.

Table 1: Parameters for adaptive disconnection (times in seconds). The cross-product of the sets of parameters was used to drive the simulations; that is, each of the 3 combinations of (α_a, β_a) and 3 combinations of (α_m, β_m) in Table (a) is used with each of the 3 sets of values in Table (b), giving 18 sets of adaptive parameters to drive the simulator.

were so identified, and the number increased to 19 with a 10-minute threshold.

How should workaholics be treated? A workaholic that would be active even if modem disconnections were unintrusive should be considered in all results, because it will limit the overall benefit available if new policies were deployed. One that is artificially busy *because* of the inconvenience of modem disconnections should be factored out.

We consulted with several users who were categorized as workaholics, about half the total number we identified. The vast majority of those consulted indicated that they ran programs to keep the connection active and avoid disconnection. Because of this specially generated activity, we have chosen to exclude workaholics from consideration in this paper unless stated otherwise.

One thing our simulations cannot tell us is whether our adaptive timeouts might turn a "regular" user into a "workaholic" by causing him or her to change behavior—either intentionally or inadvertently. Only direct empirical observation of user behavior on our prototype would provide that knowledge; this is future work.

We consider workaholics further in Section 6.4.

6 Results

We present results that compare adaptive and fixed thresholds, given a fixed definition of what constitutes a bump and aggregating across all users. We then compare these results to a set of simulations with a more conservative definition of a bump. Next we consider the effect of increasing the maximum number of prediction errors. We then include the effect of workaholics, and finally we look at variations of the adaptive thresholds among two individual users.

6.1 Adaptive versus Fixed Thresholds

From the standpoint of user annoyance, the number of times the user must wait for a new connection is of great importance. However, the longer since the user last communicated, the more willing the user is to wait to reinitiate contact. Different users may have varying levels of willingness to suffer an initial delay. As a baseline, we consider a case where a disconnect is considered a bump if the modem is not idle for at least 5 minutes after the disconnection. We vary this threshold in the next subsection.

The first set of figures shows the cumulative effect across the entire user population of approximately 100 modems, but excluding the activity of 15 workaholics. Figure 2 shows the total number of bumps encountered, on a log scale, by comparison to the total connect time. Connect time is itself relative to the off-line optimal case, as discussed above, which appears on the bottom axis at the relative value of 1. The fixed 2-minute timeout uses slightly less connect time but encounters an unacceptable total of over 20,000 bumps over a one-week period. The other fixed points encounter far fewer bumps, but only the 15-minute threshold encounters fewer bumps than the various adaptive algorithms. Compared to the 5-minute threshold, some of the adaptive algorithms encounter close to half an order of magnitude fewer bumps at no cost in connect time, while the rest reduce the bumps further in exchange for more connect time. Compared to the 10-minute threshold, virtually all the adaptive points are both below and to the left, i.e., encounter both fewer bumps and less connect time.

The fixed 15-minute threshold is an interesting case. Compared to the adaptive point that is farthest to the bottom-right of the graph, it uses 9% more connect time and encounters 27% fewer bumps. The relative weights

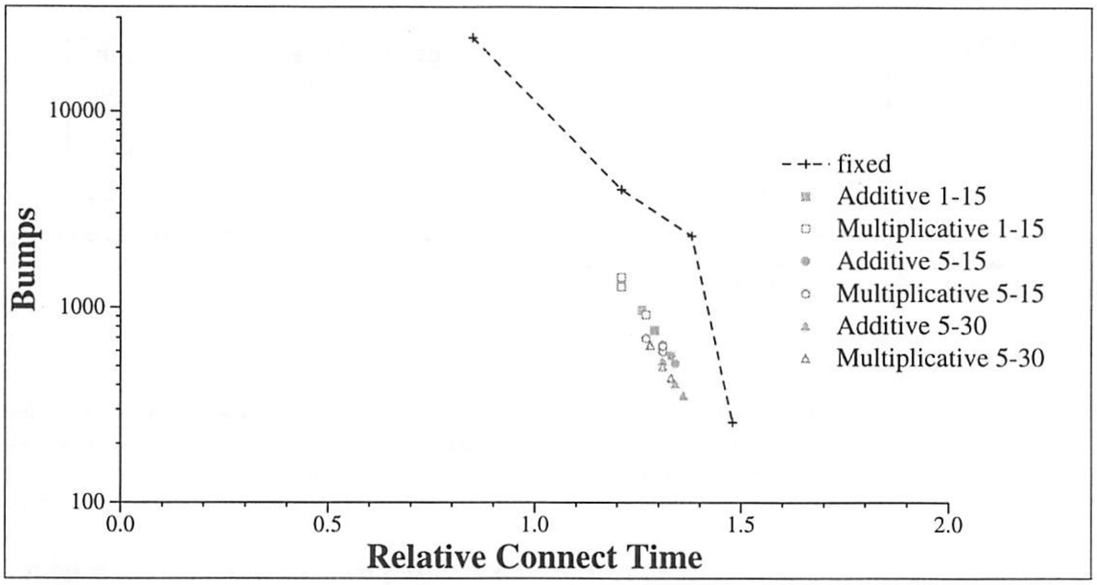


Figure 2: Total bump count, 5-minute bump threshold, excluding 15 workaholics. Note that the y-axis is on a log scale with a minimum of 100. *The optimal policy, not shown, would encounter 0 bumps for a relative connect time of 1.*

of the need to reclaim modems and the desire not to inconvenience the user would determine whether the simpler fixed threshold would be more desirable.

Focusing on the differences between additive and multiplicative thresholds, we see that the additive policies within a range of values are consistently below and to the right of the multiplicative ones. (Refer to the solid polygons of a particular shape, compared to the open ones.) This means that the additive policies are connected longer but encounter fewer bumps. This is unsurprising since the multiplicative ones react to bumps more aggressively; the same phenomenon was noted in the domain of spinning down a disk [2].

"Bump severity" weights the bumps by the extent to which they missed the threshold. Figure 3 shows the severity-versus-connection plot comparable to Figure 2. Most of the same comments apply, but the adaptive points with the lowest severity are closer to the 15-minute fixed threshold. Here, the fixed threshold reduces the severity by just 10% in exchange for the same 9% increase in connect time.

Figure 4 plots the average number of modems in use at each 30-second collection point, relative to the optimal, against the same "relative connect time" in the preceding figures. As one would expect, the more total time used by an algorithm, the more likely additional modems will be in use in parallel. The 15-minute fixed threshold uses 23% more modems on average than the optimal, whereas the adaptive algorithm with the fewest bumps uses just 13% more, equivalent to a reduction of 8% from the 15-minute threshold. The 2-minute threshold uses much less than the off-line optimal, but of course it encounters too many bumps to be of practical interest.

Figure 5 is similar to Figure 4, but with the maximum number of simultaneous modems instead of the average. With this trace and configurations, virtually all the thresholds except the fixed 2-minute one hit the same maximum. The 2-minute threshold had a lower maximum, as one would expect. The 15-minute and a couple of the adaptive thresholds, including the one that is closest to the 15-minute threshold in bumps, had a maximum that was one modem greater than the other thresholds.

6.2 Varying the Bump Threshold

Figures 6–8 present simulation results corresponding to Figures 2–4, but with a bump if the idle period is less than 10 minutes (rather than 5).[1] Note that although most numbers are directly comparable because they are relative to the optimal policy for that definition of a bump, the total counts of bumps are absolute counts. Since they exclude different numbers of workaholics, they are not directly comparable, and one should instead consider the trends within each graph.

Focusing just on the number of bumps (Figure 6, compared with Figure 2), the overall shape of the

[1]The figure corresponding to Figure 5 is omitted due to space limitations, but is virtually the same.

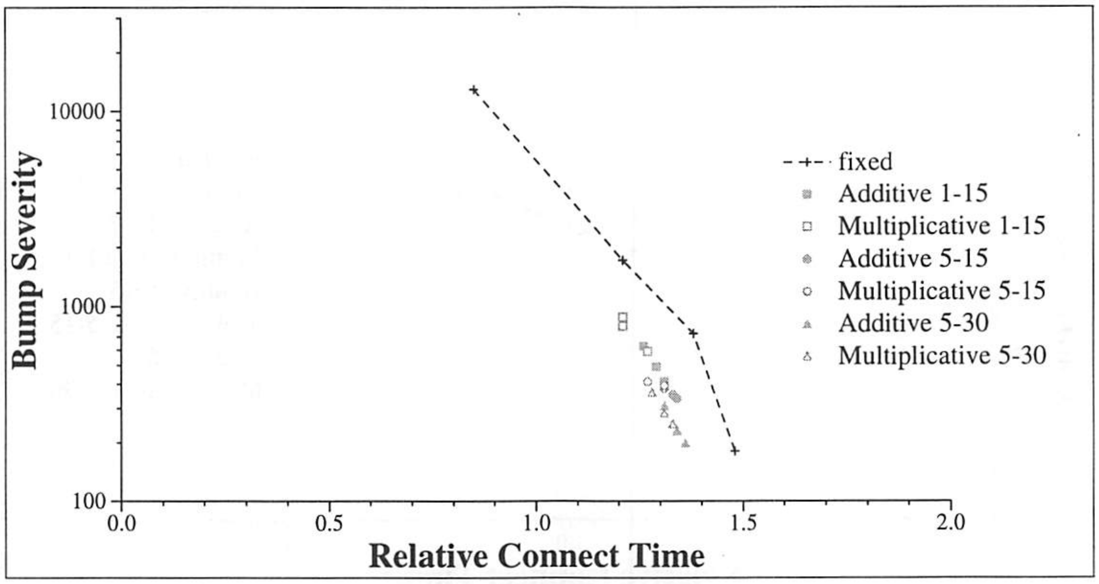


Figure 3: Bump severity, 5-minute bump threshold, excluding 15 workaholics. Note that the y-axis is on a log scale with a minimum of 100. *The optimal policy, not shown, would encounter 0 bumps for a relative connect time of 1.*

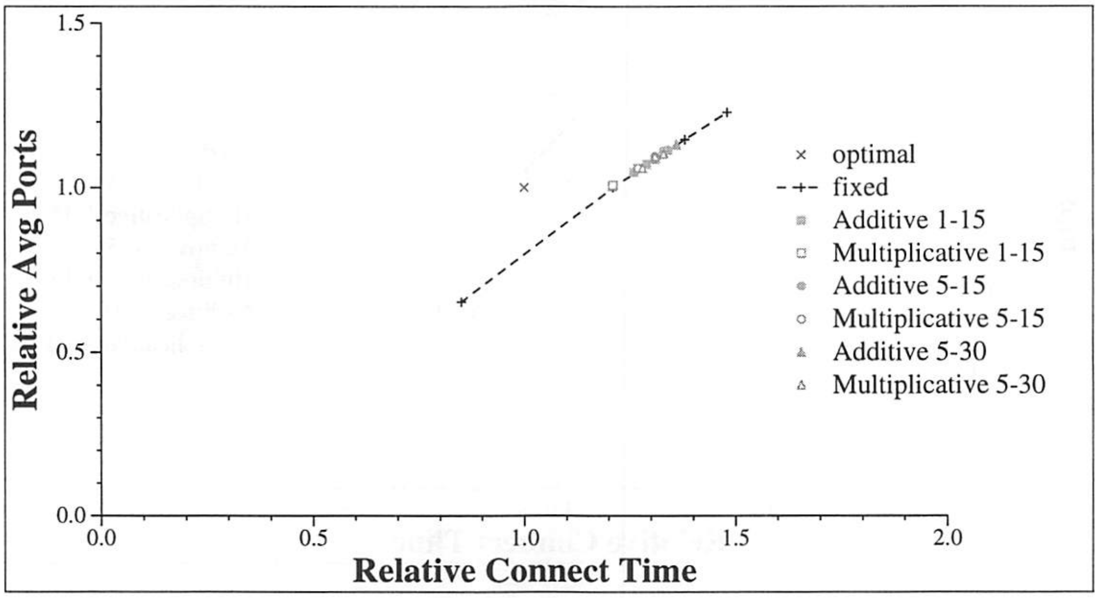


Figure 4: Mean modem usage, 5-minute bump threshold, excluding 15 workaholics.

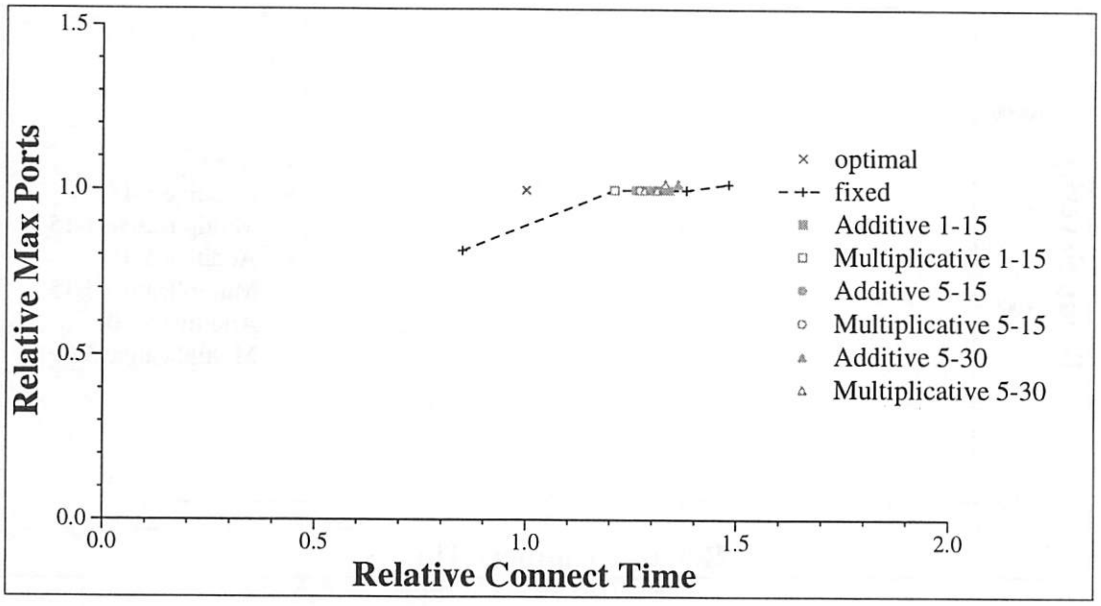


Figure 5: Maximum modem usage, 5-minute bump threshold, excluding 15 workaholics.

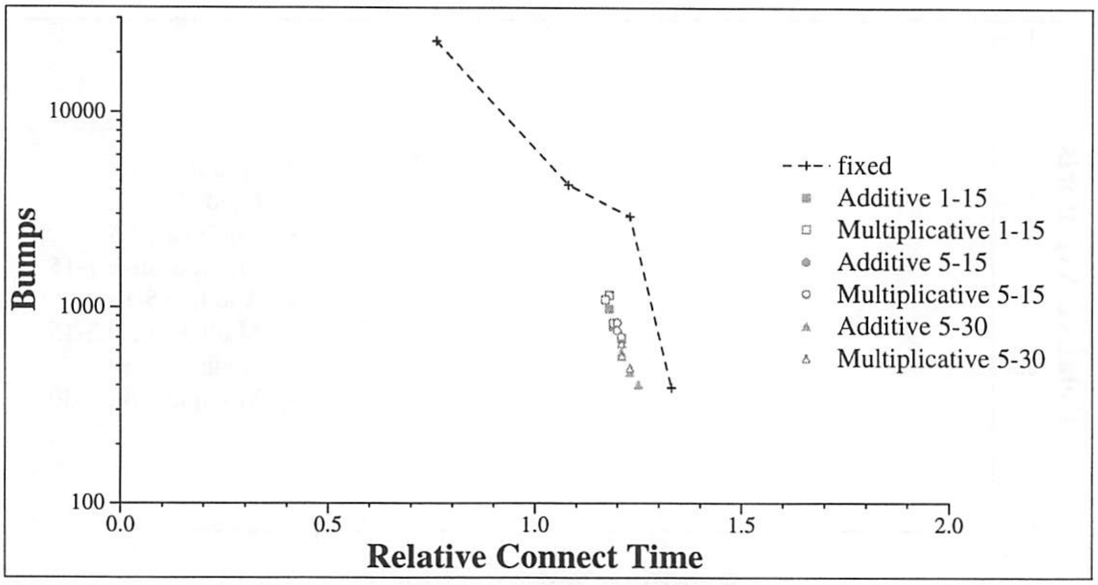


Figure 6: Total bump count, 10-minute bump threshold, excluding 19 workaholics. Note that the y-axis is on a log scale with a minimum of 100. *The optimal policy, not shown, would encounter 0 bumps for a relative connect time of 1.*

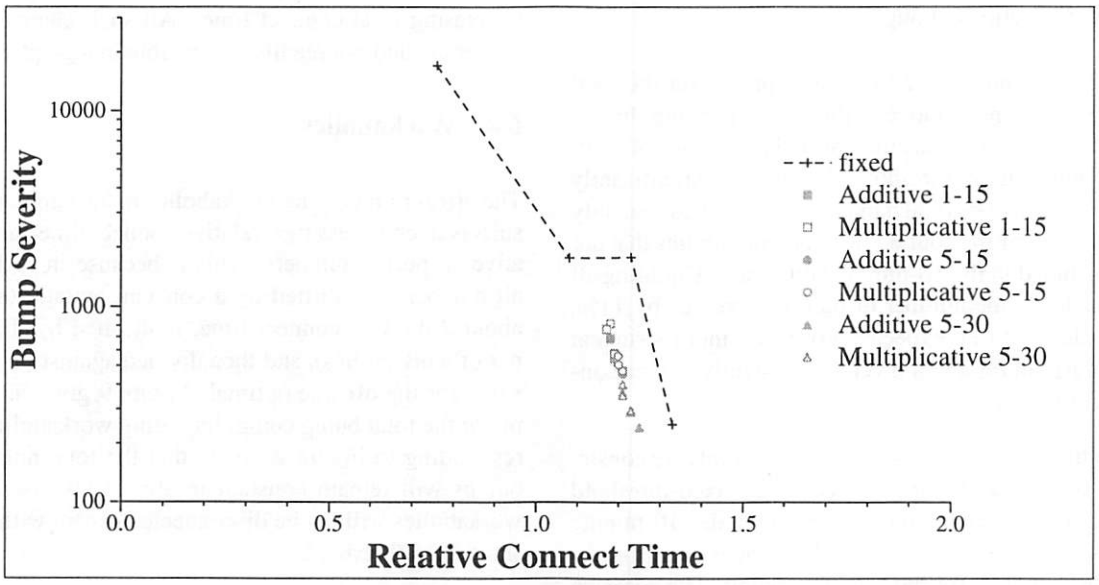


Figure 7: Bump severity, 10-minute bump threshold, excluding 19 workaholics. Note that the y-axis is on a log scale with a minimum of 100. *The optimal policy, not shown, would encounter 0 bumps for a relative connect time of 1.*

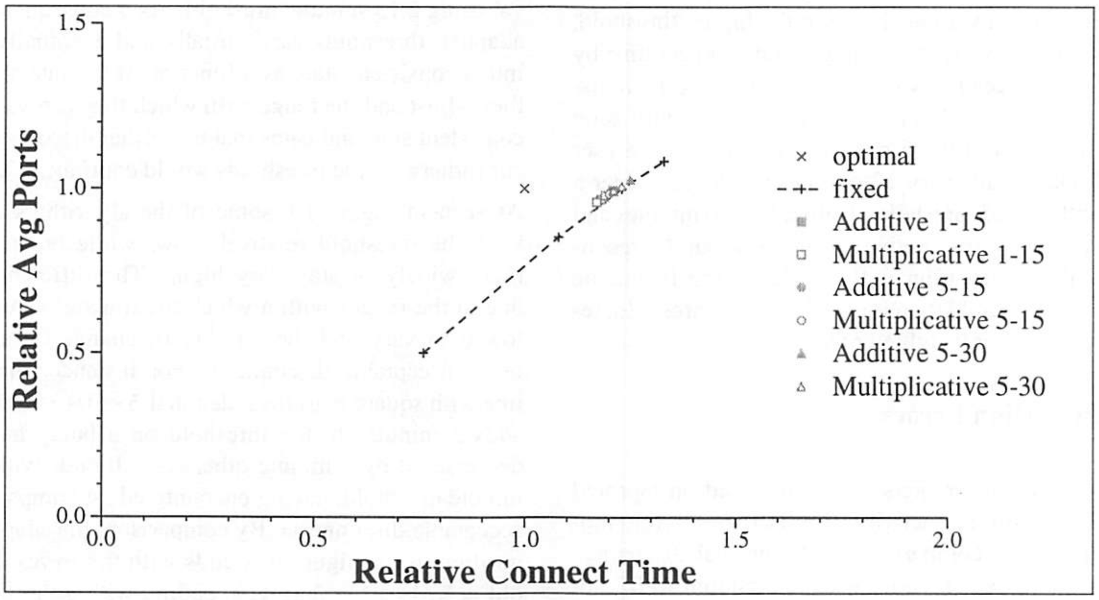


Figure 8: Mean modem usage, 10-minute bump threshold, excluding 19 workaholics.

fixed-threshold curves and the relative positions of the adaptive-threshold points are similar. There are some notable distinctions, though:

- The 2-minute fixed timeout improves on the total connect time, relative to the 5-minute bump threshold. This is because the optimal in the case of a 10-minute bump threshold disconnects significantly less often. The total bump count decreases, but only because of the four additional workaholics that are excluded in the 10-minute bump case. Counting all modems, the number of bumps increases by 11%, which is to be expected given the more stringent threshold for a bump yet a consistently low disconnect threshold.
- While in both figures the adaptive points are consistently below the line formed by the fixed-threshold points, the best adaptive point in the 10-minute bump case is closer to the 15-minute fixed threshold case than for the 5-minute bump. The adaptive policy with the fewest bumps increases the total bump count by less than 4% compared to the 15-minute fixed threshold, and reduces connect time by 6%. When bump severity is considered, the adaptive policy actually reduces overall severity by 4% compared to the 15-minute threshold.

The severity for the fixed 5-minute (Figure 3) and 10-minute (Figure 7) thresholds is virtually identical: it actually increases by about 1% with the higher threshold, while simultaneously increasing overall connect time by 14%. This is because waiting 10 minutes and then disconnecting is more likely to hit network activity soon thereafter than waiting 5 minutes. For instance, a user who checked mail every 12 minutes would encounter a bump with a fixed threshold of either 5 or 10 minutes and a need to be idle for another 10 minutes, but the reconnect would occur earlier in the cycle for the 10-minute disconnect threshold than for the 5-minute threshold, resulting in a higher weighted severity.

6.3 Prediction Errors

When our system predicts a timeout based on repeated intervals of similar activity, it disconnects the modem quickly by comparison to the normal disconnection threshold, which varies using a "random walk" approach. As a result, if the prediction is in error, the user is likely to be much more inconvenienced than with the adaptive threshold. Currently, the system counts the number of such errors for each modem and stops making predictions for a modem that has already encountered two such errors. We investigated the sensitivity to this threshold by increasing it from 2 to 5. As one might expect, the effect was to increase the bump count while decreasing total connect time. All such changes were moderate, and not readily discernible in a graph.

6.4 Workaholics

The effect of including workaholics in the simulation results is to compress the "relative connect time" and "relative ... ports" numbers. This is because in each case, all numbers get shifted by a constant amount (such as about 7 days of connect time, multiplied by the number of workaholics), and then divided against the higher value for the off-line optimal. Figure 9 gives an example of the total bump count, including workaholics, corresponding to Figure 2. Note that the total number of bumps will remain constant in almost all cases, since workaholics will not be disconnected except with a very short fixed threshold.

6.5 Threshold Variation

Figures 10 and 11 plot the variation in threshold for a subset of the different algorithms for two clients.[2] The modem in Figure 10 belonged to one of the authors and showed a marked fluctuation over time. The modem in Figure 11 was one of the workaholics, accumulating 6.8 days' worth of connect time over the 7-day trace interval using a 15-minute threshold. As a consequence, the adaptive thresholds vary initially and eventually settle into a consistent state as a function of the rate at which they adjust and the range with which they can vary. The consistent state indicates that no further disconnects occur (otherwise the thresholds would continue to adjust).

As seen in Figure 10, some of the algorithms tend to keep the threshold relatively low, while others range more widely or stay vary high. The differences are due to the ranges within which the timeout value is allowed to vary and the rapidity of change on a bump or an acceptable disconnect. For instance, the black line with square markings, denoted `5-60+300m5M30`, adds 5 minutes to the threshold on a bump and only decreases it by 1 minute otherwise. It ends with a 29-minute threshold, having encountered 24 bumps and 60 acceptable disconnects. By comparison, the adaptive algorithm in this figure that ends with the lowest threshold is `5/1.1*1.2m1M15`, ending with a threshold of 2.4 minutes and 79 bumps against 164 acceptable disconnects. The fraction of disconnects that was deemed

[2]Color versions of these two graphs, along with corresponding graphs for all algorithms rather than the selected subset, are available at `http://www.research.att.com/~douglis/modem-timeouts/graphs/`.

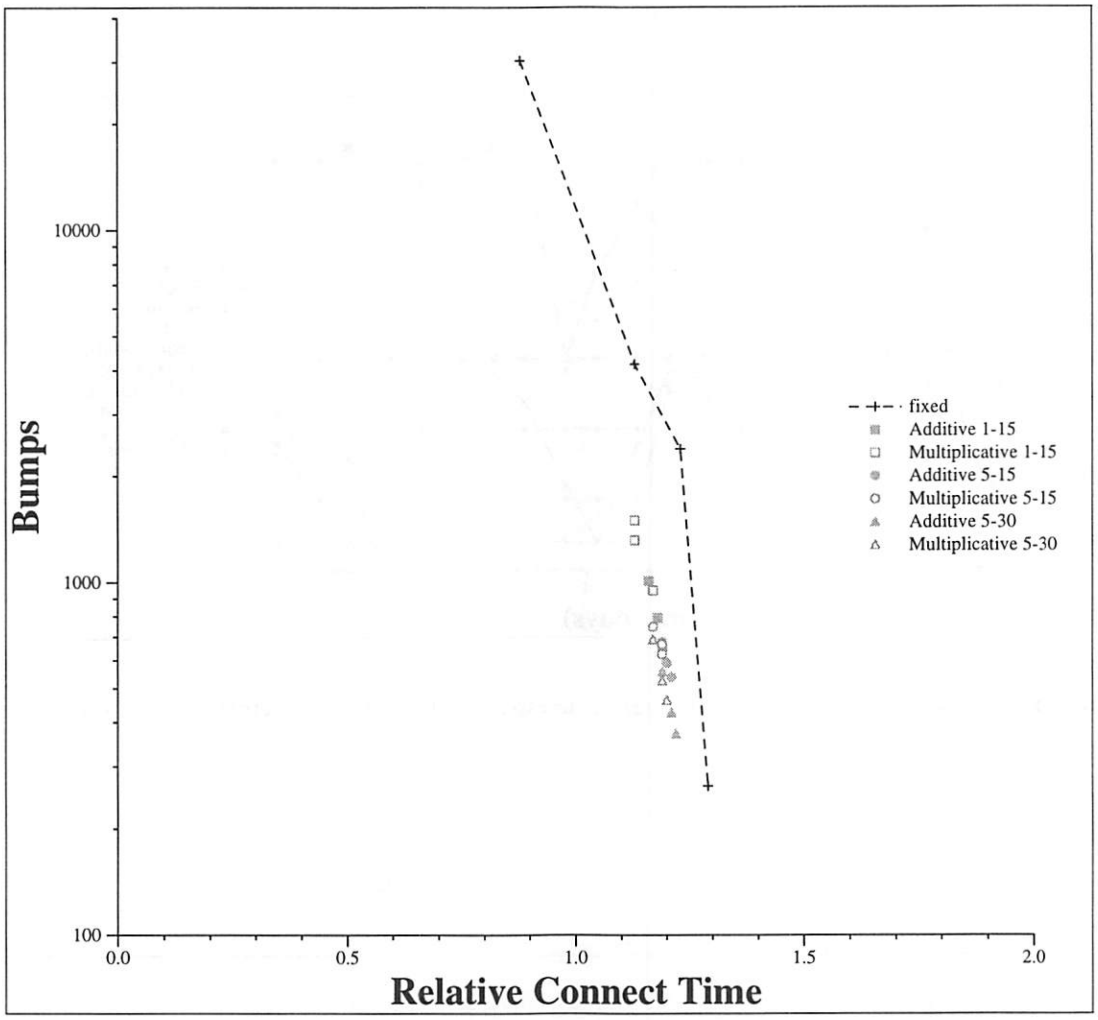


Figure 9: Total bump count, 5-minute bump threshold, including the 15 workaholics. Note that the y-axis is on a log scale with a minimum of 100. *The optimal policy, not shown, would encounter 0 bumps for a relative connect time of 1.*

unacceptable was only marginally worse in the latter case, 33% rather than 29%, but the total number of bumps in the same one-week period increased by more than a factor of three.

The variation in threshold show in Figure 11 is largely relevant in showing how long it gets to the steady state. The multiplicative algorithms generally took 2–3 days to settle to a fixed value, while the additive ones adjusted more quickly.

7 Summary and Future Work

Adaptive techniques that use past history to predict a good timeout interval for modem disconnection can reduce overall resource consumption with little or no additional inconvenience to the user. These techniques require minimal state on the part of the ISP and are easy to implement.

Using a one-week trace from our corporate environment, we found that adaptive policies can reduce cumulative connection times and average simultaneous usage by about 10–20% compared to a conservative fixed threshold, in exchange for a moderate increase in the number of disconnections that inconvenience the user.

The exact choice of which modifiers to use, and limitations to place on the range among which the timeout can vary, is thus far an imprecise art. Additional experimentation with "live users" will be helpful in further evaluating the technique and providing guidelines for these parameters. Also, these techniques must be applied to a wider user community, such as "home" users rather than "corporate" ones.

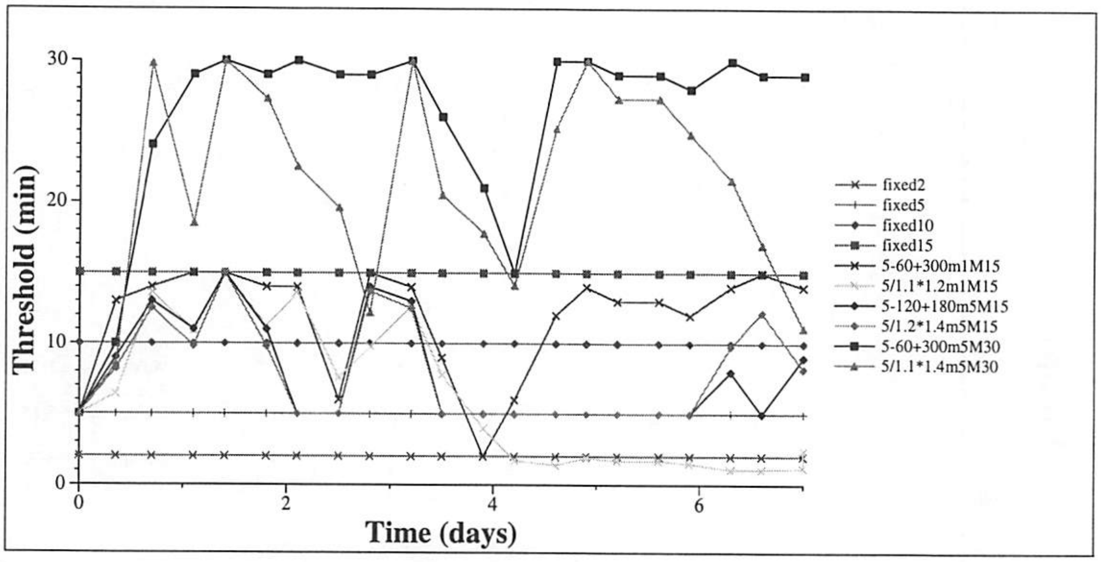


Figure 10: Variation in thresholds for a user with irregular accesses. For clarity, only a subset of the tested algorithms is included.

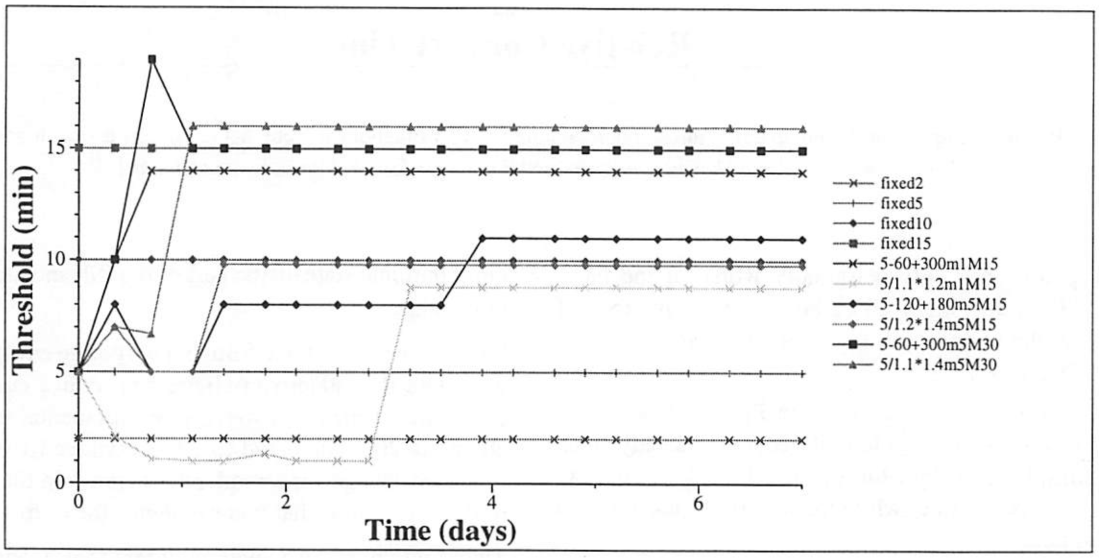


Figure 11: Variation in thresholds for a user with regular and frequent accesses (a "workaholic"). For clarity, only a subset of the tested algorithms is included.

Acknowledgments

We thank the anonymous referees for their comments. In addition, Lisa Bahler, Dan Duchamp, and Xiaoning Zhang provided helpful comments on earlier drafts, and Lorinda Cherry and S. Keshav provided additional comments on the problem.

References

[1] Daniel Barbará and Tomasz Imieliński. Sleepers and workaholics: Caching strategies in mobile environments. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the International Conference on Management of Data*, pages 1–12, New York, NY, USA, May 1994. ACM Press.

[2] Fred Douglis, P. Krishnan, and Brian Bershad. Adaptive disk spin-down policies for mobile computers. *Computing Systems*, 8(4):381–413, Fall 1995. An earlier version appeared in *Proceedings of the Second Symposium on Mobile and Location-independent Computing*, pp. 121–137, April 1995.

[3] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the Power Hungry Disk. In *Proceedings of 1994 Winter USENIX Conference*, pages 293–306, San Francisco, CA, January 1994.

[4] R. Droms. RFC 2131: Dynamic host configuration protocol, April 1997.

[5] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 41–55. Association for Computing Machinery SIGOPS, October 1991.

[6] S. Keshav, C. Lund, S. J. Phillips, N. Reingold, and H. Suran. An empirical evaluation of virtual circuit holding time policies in IP-over-ATM networks. *IEEE Journal on Selected Areas in Communications*, 13(8):1371–1382, October 1995.

[7] Joseph R. Kiniry and Christopher Metz. Cable modems: Cable TV delivers the internet. *IEEE Internet Computing*, 2(3):12–15, May–June 1998.

[8] Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *Proceedings of the 1994 Winter USENIX*, pages 279–291, San Francisco, CA, 1994.

[9] William Simpson et al. RFC 1661: The point-to-point protocol (PPP), July 1994.

[10] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 13–23. USENIX Association, November 1994.

Operation-based Update Propagation in a Mobile File System *

Yui-Wah Lee Kwong-Sak Leung
The Chinese University of Hong Kong
{clement,ksleung}@cse.cuhk.edu.hk

Mahadev Satyanarayanan
Carnegie Mellon University
satya+@cs.cmu.edu

Abstract

In this paper we describe a technique called *operation-based update propagation* for efficiently transmitting updates to large files that have been modified on a weakly connected client of a distributed file system. In this technique, modifications are captured above the file-system layer at the client, shipped to a surrogate client that is strongly connected to a server, re-executed at the surrogate, and the resulting files transmitted from the surrogate to the server. If re-execution fails to produce a file identical to the original, the system falls back to shipping the file from the client over the slow network. We have implemented a prototype of this mechanism in the Coda File System on Linux, and demonstrated performance improvements ranging from 40 percents to nearly three orders of magnitude in reduced network traffic and elapsed time. We also found a novel use of forward error correction in this context.

1 Introduction

The use of a distributed file system on a mobile client is often hindered by poor network connectivity. Although *disconnected operation* [8] is feasible, a mobile client with an extensive amount of updates should not defer propagating them to a server for too long. Damage, theft or destruction of the client before update propagation will result in loss of those updates. Further, their timely propagation may be critical to successful collaboration and in reducing the likelihood of update conflicts.

Propagation of updates from a mobile client is often impeded by *weak connectivity* in the form of wireless or wired networks that are low-bandwidth or intermittent. Aggressive update propagation under these conditions increases the demand on scarce bandwidth. A major component to bandwidth demand is the shipping of large files in their entirety. Two obvious solutions to the problem are *delta shipping* and *data compression*. The former tries to ship only the incremental differences between versions of files, and the latter "compresses out" the redundancies of files before shipping the files. Unfortunately, as discussed later in this paper, both of these methods have shortcomings that limit their usefulness.

In this paper, we focus on a radically different solution, which is called *operation-based update propagation* (or *operation shipping* for brevity). It is motivated by two observations. First, large files are often created or modified by *user operations* that can be easily intercepted and compactly represented. Second, the cost of shipping and re-executing the user operations is often significantly smaller than that of shipping the large files over a weak network.

To propagate a file, the client ships the operation to a *surrogate client* that is well connected to the server, as shown in Figure 1. The surrogate re-executes the operation, validates that the re-generated file is identical to the original, and then propagates the file via its high-speed connection to the server. If the file re-generated by the surrogate does not match that from the original execution, the system falls back to shipping the original from the client to the server over the slow connection. This validation and fall-back mechanism is essential to ensure the *correctness* of update propagation.

*This research was partially supported by the Defense Advanced Research Projects Agency (DARPA), Air Force Materiel Command, USAF under agreement number F19628-96-C-0061, the Intel Corporation, and the Novell Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, Intel, Novell, or the U.S. Government.

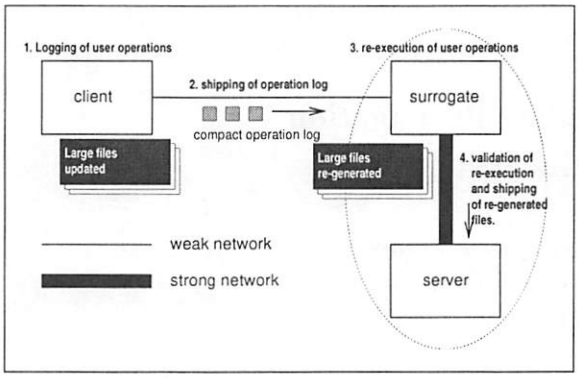

1. Logging of user operations
3. re-execution of user operations
client
2. shipping of operation log
surrogate
compact operation log
Large files updated
Large files re-generated
4. validation of re-execution and shipping of re-generated files.
weak network
strong network
server

Figure 1: An overview of operation shipping

The use of a surrogate in our approach ensures that server scalability and security are preserved — servers are not required to execute potentially malicious and long-running code, nor are they required to instantiate the execution environments of the numerous clients they service. In our model, we assume each mobile client has pre-arranged for a suitable surrogate of an appropriate machine type, at an adequate level of security, possessing suitable authentication tickets, and providing an appropriate execution environment. This assumption is reasonable, since many users of mobile clients also own powerful and well connected personal desktops in their offices, and they can pre-arrange for these otherwise idle personal machines as the surrogates.

Even though the idea of operation shipping is conceptually simple, there are many details that have to be addressed to make it work in practice. In the rest of this paper, we describe how we handle these details by implementing a prototype based on the Coda File System. Our experiment confirms the substantial performance gains of this approach. Depending on the metric and the specific application, we have observed improvements ranging from a factor of three to nearly three orders of magnitude.

2 Coda background

We have implemented our prototype by extending the Coda File System [3]. Although our experience is based on Coda, the general principles should also be applicable to other mobile file systems. We briefly describe Coda in this section; more information is available in the literature [3, 8, 7, 14, 13].

The Coda model is that there are many clients and a few servers. On each client, a cache manager, called *Venus*, carefully manages and persistently stores cached file-system objects. To support mobile computing, Coda clients can be used in *disconnected* and *weakly connected* mode, where Venus emulates the servers and allows file-system operations on cached objects. Updates are applied immediately to locally cached objects, and are also logged in a *client-modify log (CML)*. The logging mechanism allows propagation of updates to servers to be deferred until a convenient time, such as when network connectivity is restored. Venus propagates these updates with a mechanism called *trickle reintegration* [14, 13]. When propagation is attempted, a prefix of the log is shipped to the server in temporal order, the size of the prefix being determined by available bandwidth.

The effect of each mutating file-system operation is represented as a record in the CML. For example, a `chmod` operation is logged as a `CHMOD` record, a `mkdir` operation is logged as a `MKDIR` record. Furthermore, if there have been some intervening `write` operations made to an `open`'ed file, a subsequent `close` operation will be logged as a `STORE` record.

Records of the type `STORE` are special, because the associated data include the contents of the files. Therefore, these records are much bigger than records of other types. `STORE` records can be as large as several kilobytes or even megabytes, whereas other records are typically smaller than a few hundred bytes. Although the contents of a file logically constitute a part of the CML, they are physically stored in a separate *container file* in the client's local file system.

Although trickle reintegration has been shown to be effective in decoupling the foreground file-system activities from the slow propagations of updates, it still suffers from an important limitation: updated files are propagated in their entirety. In other words, although the users perceive a fast response time from the file system, the actual propagations of the updates are very slow, and they generate heavy network traffic. On a weak network, we need a more efficient scheme for shipping updates.

3 Design and Implementation

3.1 Overview

We organize our discussion according to the four steps shown in Figure 1:

1. **Logging of user operations.** When a file is updated on a client, the client keeps the new file value V (the contents of the file) and also logs the user operation O.

2. **Shipping of operation log.** If the network bandwidth is low, the client does not ship V to the server. Instead, it ships O, the fingerprint of V, and other meta-data to a *surrogate* client.

3. **Re-execution of operations.** The surrogate re-executes O and produces a file value V'.

4. **Validation of re-execution.** The surrogate validates the re-execution by checking the fingerprints and meta-data. If it accepts the re-execution, then it will present V' to the server; otherwise, it will report failure to the client, which will then fall back to value shipping and ship V directly to the server.

3.2 Logging of user operations

3.2.1 Level of abstraction

We need to find the right level of abstraction for logging operations, such that the operation logs are compact. Currently, Venus logs only the low-level *file-system operations*. File-system operation logs are not compact enough for efficient operation shipping, since they contain the contents of the files.

On the other hand, computer users tend to think at a higher level. Typically, when they are working with a computer, they issue a series of *commands* to it. Since human beings type relatively slowly, the commands must be compact; hence, we focus on this level for operation shipping. When we speak of *user operations*, we mean the high level commands of computer users that can be intercepted, logged, and replayed later.

There must be an entity that intercepts the user operations and supplies the relevant information to the file system. This entity can be an interactive shell, or it can be the application itself. We refer to the two cases as *application-transparent* and *application-aware* logging.

3.2.2 Application-transparent logging

Application-transparent logging is possible if the application is *non-interactive*. In this case, an interactive shell can intercept the information related to the user operation. For example, operation logging can be transparent to the following types of applications (examples are listed in parentheses): compiler and linkers (`gcc, yacc,` and `ld`), text processors (`latex` and `troff`), file-format converters (`dvips` and `sgml2latex`), software-project management tools (`make`), and file packagers (`tar`).

For application-transparent operation shipping, we need not modify the applications, we simply need to modify some interactive shells – the meta-applications – such as `bash` and `tcsh`. Fortunately, there are much fewer meta-applications than applications. We assume, of course, that the users are willing to use a modified shell to take advantage of operation shipping.

We extend the file-system API (application-programmer interface) with two new commands for the `ioctl` system call: `VIOC_BEGIN_OP` and `VIOC_END_OP`. They delineate the beginning and the end of a user operation. The user operation is tagged with the process group , so forked children in the same group are considered part of the same user operation. When the file system receives a file-system call, it can determine whether the call comes from a tagged user operation by examining the process-group information of the caller. The information necessary for re-execution, including the name of the user command, the command-line arguments, current working directory, and so on, is passed to the file system with the `VIOC_BEGIN_OP` command.

Our file system provides the logging mechanism, but the logging entities can choose the appropriate logging policies. For example, an interactive shell may allow users to specifically enable or disable operation logging based on certain criteria.

We have experimented with the `bash` shell, a common Unix shell. We added a few lines of code so that the modified shell issues the `VIOC_BEGIN_OP` and `VIOC_END_OP` commands appropriately. Currently, we implement the most straightforward policy: the shell logs every user operation. In the future, we may implement a more flexible policy.

3.2.3 Application-aware logging

Application-aware logging is needed when the application is interactive. In this case, the application is involved in capturing the user operations. The following types of applications fall into this category (examples are listed in parentheses): text editors and word processors (`emacs, vi, Applix Word` and `Microsoft Word`), drawing tools (`xfig` and `Corel Draw`), presentation software (`Applix Present` and `Microsoft PowerPoint`), computer-aided-design tools (AutoCAD and magic), and image manipulators (xv and GNU Image Manipulation Program).

In this paper, we focus on application-transparent operation shipping. The mechanism that we have designed, and the evaluation that we have performed, is limited to non-interactive applications. We plan to study application-aware operation shipping next, and we will report our findings in the future.

3.3 Shipping of operation log

3.3.1 Shipping mechanism

The *reintegrator*, which is a user-level thread within Venus, manages update propagation. It periodically selects several records from the head of the CML and ships them to the servers. For records with no user-operation information attached, the reintegrator uses value shipping and makes a ViceReintegrate remote procedure call (RPC) to the server. The server, when processing the RPC, *back-fetches* the related container files from the client. If the reply of the RPC indicates success, the reintegrator will locally commit the updates. Local commitment of updates is the final step of successful update propagation, and includes updating the states of relevant objects, such as version vectors and dirty bits, and truncating the CML.

If user-operation information is available for a record, the reintegrator will attempt operation shipping first. All the records associated with the same user operation will be operation shipped altogether. The reintegrator selects the records, packs the operation log, and makes a UserOpPropagate RPC to the surrogate. If the reply indicates success, the reintegrator will locally commit the updates. However, if the reply indicates failure, the reintegrator will set a flag in each of the records indicating that it has tried and failed propagation by operation shipping. These records will then be value shipped.

3.3.2 Cost model

The current version of our prototype attempts operation shipping for a record whenever there is user-operation information available. This *static* approach implicitly assumes that the connectivity between a mobile client and its servers is always weak. In real life, a mobile client may have strong connectivity occasionally. During that time, as explained in the following paragraphs, value shipping is more efficient than operation shipping. We plan to enhance our prototype so that mobile clients *dynamically* decide whether they should use operation shipping or value shipping.

Our cost model compares the costs of value shipping with that of operation shipping. For each case, there are two different costs involved: network traffic and elapsed time.

For value shipping, assuming the overhead is negligible, the network traffic is the total length L of the updated files, and the elapsed time is $T_v = L/B_c$, where B_c is the bandwidth of the network connecting the client to the server.

For operation shipping, the network traffic is the length of the operation log, L_{op}, and the elapsed time is T_{op}. The latter is composed of four components: (1) the time needed to ship the operation log (L_{op}/B_c), (2) the time needed for re-executing the operation (E), (3) the time needed for additional computational overhead (H_{op}) such as computing checksum information and encoding and decoding of forward-error-correction codes, and (4) the time needed to ship the updated files to the servers. There are two cases for the last component. If the re-execution passes the validation (accepted), the updated files will be shipped from the surrogate (the time cost will be L/B_s, where B_s is the bandwidth of the network connecting the surrogate to the server); if the re-execution fails the validation, the updated file will be shipped from the client (the time cost will be L/B_c). The following equation summarizes the time costs involved:

$$T_{op} = \begin{cases} L_{op}/B_c + E + H_{op} + L/B_s & \text{if accepted} \\ L_{op}/B_c + E + H_{op} + L/B_c & \text{if rejected} \end{cases} \quad (1)$$

Therefore, operation shipping is more favorable than value shipping only in certain condition. Operation shipping saves network traffic if the operation log is more compact than the updated files ($L_{op} < L$). Also, it speeds up the update propagation ($T_{op} < T_v$) if the following five conditions are true: (1) the re-execution is accepted, (2) the operation log is compact ($L_{op} \ll L$), (3) the re-execution is fast ($E \ll L/B_c$), (4) the time needed for additional computational overheads is small ($H_{op} \ll L/B_c$), and (5) the surrogate has a much better network connectivity than the client ($B_s \gg B_c$).

3.4 Re-execution of user operations

3.4.1 Re-execution mechanism

Although we anticipate most re-executions will be successful (execute completely and pass the validation), we have to prepare for the possibility that they may fail (cannot execute completely or fail the validation). Therefore, we have to ensure that re-executions are abortable transactions such that failed re-executions will have no lasting effect. We implement this as follows.

Upon receiving a `UserOpPropagate` RPC from the client, Venus on the surrogate will temporarily put the affected volume [1] into *write-disconnected* mode, and then re-executes the user operation via a spawned child called the *re-executor*. During the re-execution, since the volume is write-disconnected, input files can be retrieved from the servers, but file-system updates are not written immediately to the server. These updates are tagged with the identity of the re-execution and are logged in the CML. If the re-execution later passes the validation, the surrogate will re-connect the volume with the servers and reintegrate the updates. At the end, and only when the reintegration succeeds, the surrogate will locally commit the updates, and indicate a success to the client in a RPC reply. On the other hand, failures may happen any time during the re-execution, the validation, or the reintegration. If any such failures occur, the surrogate will discard all the updates and indicate a failure to the client in a RPC reply.

It is possible that a reintegration completes successfully at the servers, but the RPC response fails to arrive at the client in spite of retransmission. This can happen when there is an untimely failure of the surrogate or the communication channels. We make use of the existing Coda mechanism of ensuring atomicity of reintegrations [13] to handle this kind of failures. In short, the client presumes an reintegration has failed if it does not receive a positive response from the surrogate, and it will retry the reintegration. At the same time, the server retains the information necessary to detect whether a record has already been reintegrated earlier. If a client attempts such an improper retry, the server will reply with the error code `EALREADY` (Operation already in progress), and the client will then know that the records have already been successfully reintegrated in a previous attempt, and it will simply locally commit those records.

3.4.2 Facilitating repeating re-executions

A re-execution of a user operation is accepted when it produces a result that is identical to that of the original execution. We say the re-execution is *repeating* the original execution. Only these repeating re-executions are useful to operation shipping. We know that a deterministic procedure will produce the same result in different runs provided that it has the same input and the same environment in each run. Our file system makes use of this principle to facilitate repeating re-executions.

[1] In Coda, a volume is a collection of files forming a partial subtree of the Coda name space.

First, the re-executor runs with the four Unix-process attributes identical to that of the original execution: (1) the working directory, (2) the environment-variable list, (3) the command-line arguments, and (4) the file-creation mask [23].

Second, our file system expects that the surrogate machine has software and hardware configurations similar to the client machine. Two machines are said to be identically configured if they have the same CPU type, the same operating system, the same system-header files, the same system libraries, and any other system environments that can affect the outcomes of computations on the two machines.

Third, if a re-execution requires an input file stored in Coda, we can rely on the file system to ensure that the client and the surrogate will use an identical version of the input file. Coda can ensure this because a client ships updates to its servers in temporal order, and the surrogate will always retrieve the latest version of a file from the servers. For example, consider a user issuing three user operations successively on a client machine: (Op1) update a source file using an editor, (Op2) compile the source file into an object file using a compiler, and (Op3) update the source file again. When the surrogate re-executes Op2, the client must have shipped Op1 but not Op3, and the re-executor will see exactly the version updated by Op1.

We emphasize that our file system does not guarantee that all re-executions will be repeating their original executions; it just increases the probability of that happening. More importantly, it ensures that only repeating re-executions will be accepted. This is achieved by the procedure of validation (Section 3.5).

In the evaluation section, we will see that many applications exhibit repeating re-executions. Although some other applications exhibit non-repeating side effects during re-executions, these side effects can be handled in simple ways. Therefore, we believe we can use operation shipping with a large number of applications.

3.4.3 Status information

In addition to updating contents of files, user operations also update some meta-data (status information). Some of the meta-data are internal to the file system and are invisible to the users (e.g., every `STORE` operation has an identity number for concurrency control); some are external and are visible to the users (e.g., the last-modification time of a file). In both cases, to make a re-execution's result identical to that of the original execution, the values of the meta-data of the re-execution should be reset to those of the original execution. Therefore, the client needs to pack these meta-data as part of the operation log, and the surrogate needs to adjust the meta-data of the re-execution to match the supplied values.

3.4.4 Non-repeating side effects

We focus on applications that perform deterministic tasks, such as compiling binaries or formatting texts, and exclude applications such as games and probabilistic search that are randomized in nature. In an early stage of this project, we expected the re-executions of these applications to repeat their original executions completely. However, we found that some common applications exhibited non-repeating side effects. So far we have found two types of such side effects: (1) time stamps in output files, and (2) temporary names of intermediate files. Fortunately, we are able to handle these side effects automatically, so we are still able to use operation shipping with these applications. We will discuss the handling methods in the two following subsections.

We also anticipate a third possible kind of side effect: external side effects. For example, if an application sends an email message as the last step of execution, then the user may be confused by the additional message sent by re-execution. To cope with this kind of side effect, we plan to allow users to disable the use of operation shipping for some applications.

3.4.5 Side effects due to time stamps

Some applications, such as `rp2gen`, `ar`, and `latex`, put time stamps into the files that they produce. `rp2gen` generates stubs routines for remote procedure calls, `ar` builds library files, and `latex` formats documents. They use time stamps for various reasons. Because of the time stamps, a file generated by a re-execution will differ slightly from the version generated by the original execution. Observing that only a few bytes are different, we can treat the changed bytes as "errors" and use the technique of forward error correction (FEC) [6, 4] to "correct" them. (We are indebted to Matt Mathis for suggesting this idea to us.)

Our file system, therefore, does the following. Venus on the remote client computes an error-correction code (we use the Reed-Solomon code) for each updated file that is to be shipped by operation, and it packs the code as a part of the operation log. Venus on the surrogate, after re-executing the operation, uses the code to correct the time stamps that may have occurred in the re-generated version of the file.

Note that this is a novel use of the Reed-Solomon code. Usually, data blocks are sent together with parity blocks (the error-correction code); but our clients send only the parity blocks. The data blocks are instead re-generated by the surrogate. Whereas others use the code to correct communication errors, we use it to correct some minor re-execution discrepancies. If a discrepancy is so large that our error-correction procedure cannot correct it, our file system simply falls back to value shipping. This entails a loss of performance but preserves correctness.

The additional network traffic due to the error correction code is quite small. We choose to use a (65535,65503) Reed-Solomon block code over $GF(2^{16})$. In other words, the symbol size is 16 bits, each block has 65,503 data symbols (131,006 bytes) and 32 parity symbols (64 bytes). The system can correct up to 16 errors (32 bytes) in each data block.

However, the additional CPU time due to encoding and decoding is not small. We discuss this in more detail in Section 4.4. Also, the Reed-Solomon code cannot correct discrepancies that change length (for example, the two timestamps "`9:17`" and "`14:49`" have different lengths). The rsync algorithm [24] can handle length change, but we favor the Reed-Solomon code because it has a smaller overhead on network traffic.

3.4.6 Side effects due to temporary files

The program `ar` is an example of an application that uses temporary files. Figure 2 shows the two CMLs on the client and the surrogate after the execution and re-execution of the following user operations:

```
ar rv libsth.a foo.o bar.o.
```

The user operation builds a library file `libsth.a` from two object modules `foo.o` and `bar.o`.

Note that `ar` used two temporary files `sta09395` and `sta16294` in the two executions. The names of the temporary files are generated based on the identity numbers of processes executing the application, and hence they are time dependent. Our validation procedure (Section 3.5) might naively reject the re-execution "because the records are different."

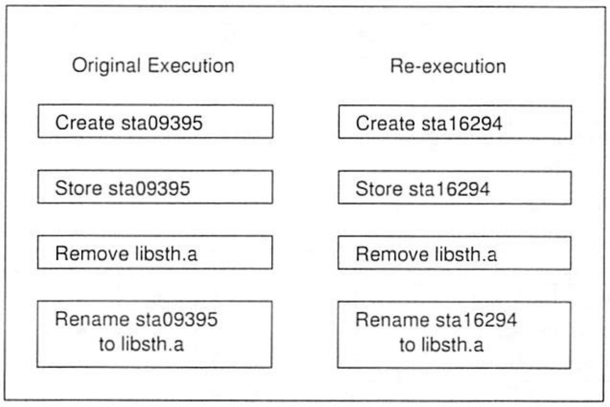


Figure 2: CMLs of two executions of `ar`

Temporary files appear only in the intermediate steps of the execution. They will either be deleted or renamed at the end, so their names do not affect the final file-system state. An application uses temporary files to provide execution atomicity. For example, `ar` writes intermediate computation results to a temporary file, and it will rename the file to the target filename only if the execution succeeds. This measure is to ensure that the target file will not be destroyed accidentally by a futile execution.

If a temporary file is created and subsequently *deleted* during the execution of a user operation, its modification records will be deleted by the existing *identity cancellation* procedure [7]. They will not appear in the two CMLs and will not cause naive rejections of re-execution.

However, if a temporary file is created and subsequently *renamed* during the execution of a user operation, its modifications records will be present in the CMLs, and might cause our validation to reject the re-execution. Our file system uses a procedure of *temporary-file renaming* to compensate for the side effect. This procedure is done after the re-executor has finished the re-execution and before the surrogate begins the validation.

The idea of the temporary-file renaming is to scan the two CMLs and identify all the temporary files as well as their final names. We identify temporary files by the fact that they are created and subsequently renamed in an user operation. For each temporary file used by the surrogate, our file system determines the temporary file name N used by the client in the original execution. It thus renames the temporary file to N. In our `ar` example, the temporary file `sta16294` will be renamed to `sta09395`.

3.5 Validation of re-executions

3.5.1 Validation mechanism

Validation is done after the handling of side effects, and the adjustments of status information. By that time, if a re-execution is repeating its original execution, the set of mutations incurred on the surrogate should be the same as that on the client. Since mutations are captured on CMLs, our file system can validate a re-execution by comparing the relevant portion of the CML of the surrogate to that of the client.

To facilitate the comparison, the client packs every record in the relevant portion of the CML as part of the operation log. However, the container files, which are associated with `STORE` records, are not packed; otherwise they would incur a heavy network traffic for shipping the operation log, amounting to the traffic needed for value shipping. Instead, the client packs the fingerprint of each container file. When comparing CMLs, the surrogate asserts that two container files are equal if they have the same fingerprint.

3.5.2 Fingerprint

A fingerprint function produces a fixed-length fingerprint $f(M)$ for a given arbitrary-length message M. A good fingerprint function should have two properties: (1) computing $f(M)$ from M is easy, and (2) the probability that another message M' gives the same fingerprint is small. For our purpose, the messages for which we find the fingerprints are the contents of the container files.

Our file system employs MD5 (Message Digest 5) fingerprints [17, 21]. Each fingerprint has 128 bits, so the overhead is very small. Also, the probability that two different container files give the same fingerprint is very small; it is in the order of $1/2^{64}$.

The fact that the probability is non-zero, albeit very small, may worry some readers. However, even value shipping is vulnerable to a small but non-zero probability of error. That is, there is a small probability that a communication error has occurred but is not detected by the error-correction subsystem of the communication channel. We believe people can tolerate the small probabilities of errors of both operation shipping and value shipping.

Test	Name	Nature	NF	Size (KB)	SE1	SE2
T1	rp2gen callback.rpc2	RPC2 stub generator	5	27.5	●	
T2	rp2gen adsrv.rpc2	RPC2 stub generator	5	76.3	●	
T3	yacc parsepdb.yacc	compiler compiling	1	23.5		
T4	c++ -c counters.cc -o counters.o	compiling	2	26.0		
T5	c++ -c pdlist.cc -o pdlist.o	compiling	2	62.4		
T6	c++ -c fso_daemon.cc -o fso_daemon.o	compiling	2	265.3		
T7	c++ parserecdump.o -o parserecdump	linking	1	23.0		
T8	ar rv libdir.a ...	library building	1	70.2	●	●
T9	ar rv libfail.a ...	library building	1	363.1	●	●
T10	tar xzvf coda-doc-4.6.5-3-ppt.tgz	extracting files	5	269.5		
T11	make coda (in coda-src/blurb)	compiling/linking	3	69.9		
T12	make coda (in coda-src/rp2gen)	compiling/linking	10	237.1		
T13	tar cvf update.tar ...	packaging files	1	60.2		
T14	sgml2latex guide.sgml	translator	1	41.8		
T15	sgml2latex rvm_manual.sgml	translator	1	270.0		
T16	latex usenix99.tex	text formatter	3	93.4	●	

We ran 16 tests using nine applications with some real-life files. For each test, we list the name, nature, the number of the files that were updated (NF), and the total size of the files. Some of the applications exhibited non-repeating side-effects due to time stamps (SE1) and temporary files (SE2), they are marked by bullet points (●) in the table.

Figure 3: Selected tests and applications

4 Evaluation

At this time, we do not sufficiently understand client usage patterns to accurately model overall performance improvement. However, we have selected a set of commonly used non-interactive applications that allow us to focus on the following three questions:

1. *Is operation shipping transparent to the applications?*
2. *What is the extent of network-traffic reduction that can be achieved by using operation shipping?*
3. *What is the extent of elapsed-time reduction that can be achieved by using operation shipping?*

We shall answer these questions in the following subsections, after we briefly describe the experimental setup.

4.1 Experimental setup

The client, the surrogate, and the server machine used in the experiments were a Pentium 90MHz, a Pentium-MMX-200MHz, and a Pentium 90MHz machine respectively. All three machines were running the Linux operating system (kernel version 2.0.35). The network between the surrogate and the server was a 10-Mbps Ethernet. The network bandwidth between the remote client and the surrogate varied in different tests, and we used the Coda failure emulation package (`libfail` and `filcon`) [18] to emulate different network bandwidths on a 10-Mbps Ethernet.

We performed 16 different tests using nine common non-interactive applications (Figure 3). We used real-life input files, found in our environment, for the tests. We selected the tests such that the data size in each test was close to one of the three reference sizes: 16, 64, and 256 Kbytes. The data size is defined as the total size of the files updated by an operation. The 16 tests were labeled as $T1, T2, \cdots, T16$. Each test was repeated three times.

4.2 Transparency to applications

We make no claim that operation shipping can be used transparently with all non-interactive applications. For example, we anticipate that operation shipping probably cannot be used with the `-j <n>` mode of `GNU Make`, which runs `n` jobs in parallel. Fortunately, so far all nine selected applications can be used transparently with operation shipping. Three of them exhibit non-repeating side effects, but these side effects can be compensated by our handling techniques.

Test	Nature	Traffic by value-shipping (Kbytes) L_v	Traffic by operation-shipping (Kbytes) L_{op}	Traffic reduction by operation-shipping L_v/L_{op}	Expected traffic by data-compression (Kbytes) $L_{v,gz}$	Expected traffic reduction by data compression $L_v/L_{v,gz}$
T1	rp2gen	28.7	2.0	14.4	6	4.8
T2	rp2gen	77.5	1.9	40.8	9.6	8.1
T3	yacc	23.7	1.0	23.7	5.4	4.4
T4	c++ -c	27.1	1.9	14.3	7.9	3.4
T5	c++ -c	63.4	1.8	35.2	17.9	3.5
T6	c++ -c	266.3	2.0	133.2	71.6	3.7
T7	c++	23.9	2.0	12.0	8.4	2.8
T8	ar	70.2	1.9	36.9	22	3.2
T9	ar	364.0	2.2	165.5	78.6	4.6
T10	tar x	271.8	4.7	57.8	71.5	3.8
T11	make	71.6	2.3	31.1	25.3	2.8
T12	make	242.0	5.9	41.0	81.8	3.0
T13	tar c	60.2	1.0	60.2	10	6.0
T14	sgml2latex	42.0	1.0	42.0	13.8	3.0
T15	sgml2latex	270.3	1.1	245.7	71.0	3.8
T16	latex	94.1	1.4	67.2	35.5	2.7

In column 5, we list the network traffic reduction factors by operation shipping L_v/L_{op}, where L_v and L_{op} is the network traffic by value shipping and by operation shipping respectively. In column 7, we also list the expected network traffic reduction factors $L_v/L_{v,gz}$ if we used data compression ($L_{v,gz}$ is the expected size of the compressed traffic).

Figure 4: Network traffic reductions by operation shipping (and by data compression)

4.3 Network traffic reduction

For each test, we measured the traffic required for propagating the update by value shipping and by operation shipping. Both the file data and the overhead are included in the traffic. In particular, for operation shipping, all fields in the operation logs: command, command-line arguments, current working directory, environment list, file-creation mask, meta-data, and fingerprints, and so on, were counted towards the traffic.

In Figure 4, we show the traffic reduction L_v/L_{op}, where L_v and L_{op} are the traffic volumes required for the update propagation by value shipping and by operation shipping respectively.

Previous Coda projects [7, 14] have shown that cancellation optimization is effective in reducing the network traffic needed for propagating updates. For example, if a file is stored several times, then only the last STORE record is needed to be shipped. When we took the measurements, we did not wait for any possible cancellation optimization to happen, therefore, the measured traffic reductions achieved by operation shipping alone represent the best-case numbers. We excluded cancellation optimization because its effectiveness depends on usage patterns, and should be studied using file-reference traces. At this stage, we do not have file-reference traces performed at the level of user operations, so we have to evaluate operation shipping in isolation of cancellation optimization.

Nevertheless, the traffic reductions achieved by operation shipping were much more substantial than that achieved by cancellation optimization. [2] In 13 out of the 16 tests, the reduction exceeded a factor of 20; the highest reduction factor was 245.7 (T15); the smallest reduction was 12 (T7). In other words, operation shipping reduced the network traffic volumes by one to nearly three orders of magnitude.

[2] In their study of file-reference traces, Mummert, Ebling and Satyanarayanan [14] have reported that about 50 % of network traffic was saved when modification records were allowed to stay in the modification log for 600 seconds, waiting for possible cancellations to happen.

			Elapsed time (msecs)					
		Data size	9.6-Kbps		28.8-Kbps		64-Kbps	
Test	Nature	(Kbytes)	T_v	T_{op}	T_v	T_{op}	T_v	T_{op}
T1	rp2gen	27.5	27,921	8,282	9,666	6,404	4,539	5,637
			(312)	(73)	(8)	(50)	(20)	(37)
T2	rp2gen	76.3	71,937	9,322	24,294	7,358	11,416	6,706
			(27)	(61)	(39)	(9)	(133)	(90)
T3	yacc	23.5	22,025	3,215	7,563	2,364	3,506	2,049
			(31)	(60)	(13)	(34)	(0)	(9)
T4	c++ -c	26.0	25,112	5,098	8,683	3,491	4,164	2,928
			(64)	(31)	(38)	(88)	(176)	(107)
T5	c++ -c	62.4	59,144	7,546	19,899	5,927	9,591	5,377
			(254)	(48)	(51)	(12)	(93)	(43)
T6	c++ -c	265.3	257,143	15,645	88,274	13,877	39,167	13,181
			(23,989)	(82)	(8,418)	(92)	(233)	(9)
T7	c++	23.0	22,218	4,425	7,637	2,874	3,599	2,297
			(27)	(27)	(12)	(30)	(12)	(20)
T8	ar	69.3	65,473	5,613	22,059	4,104	10,571	3,646
			(58)	(21)	(77)	(25)	(343)	(138)
T9	ar	363.1	345,241	13,143	118,929	11,634	55,725	10,944
			(24,172)	(142)	(7,402)	(74)	(3,472)	(91)
T10	tar x	269.5	247,674	12,825	85,041	9,448	39,954	8,448
			(327)	(156)	(276)	(96)	(182)	(60)
T11	make	69.9	67,113	8,839	22,723	6,793	10,633	6,115
			(580)	(79)	(354)	(25)	(142)	(30)
T12	make	237.1	224,135	22,272	77,279	18,098	36,256	17,085
			(2,452)	(132)	(73)	(39)	(293)	(396)
T13	tar c	60.0	55,355	3,674	18,826	2,978	8,802	2,602
			(36)	(8)	(33)	(92)	(7)	(74)
T14	sgml2latex	41.8	38,602	5,433	13,160	4,648	6,209	4,439
			(15)	(18)	(12)	(25)	(87)	(235)
T15	sgml2latex	270.0	245,709	13,780	83757	12852	39414	12600
			(266)	(103)	(162)	(42)	(32)	(107)
T16	latex	93.4	86,619	8,522	29,429	7,194	13,869	6,243
			(30)	(646)	(54)	(669)	(49)	(39)

Elapsed time, in milliseconds, for update propagation using value shipping (T_v) and operation shipping (T_{op}) under three different network conditions. Figures in parentheses are standard deviations from three runs.

Figure 5: Elapsed time for value shipping and operation shipping.

4.4 Reduction of elapsed time

We also measured the elapsed time for propagating an update by value shipping and by operation shipping. The elapsed time is the time to complete the respective remote procedure calls: `ViceReintegrate` for value shipping, and `UserOpPropagate` for operation shipping. For the latter, the elapsed time comprises the time for shipping the operation log, re-executing the operation, and other overhead, such as checking the fingerprints. Since the elapsed time depends heavily on the network bandwidth, we measured it under three different network bandwidths: 9.6, 28.8, and 64.0 kilobits per second. The measurements are shown in Figure 5.

We summarize the speedups for the tests in Figure 6. The speedup is defined to be the ratio T_v/T_{op}, where T_v and T_{op} are the elapsed time for value shipping and operation shipping respectively.

We found that the speedups were substantial. They were the most substantial in the 9.6-Kbps network. Eight out of the 16 tests were accelerated by a factor exceeding 10. The maximum speedup was 26.3 (T9); the minimum speedup was 3.4 (T1). In the other two networks, the speedups ranged from a factor of 1.4 to 10.2. (There was one exception: test T1 slowed down when using operation shipping at 64 Kbps.)

However, we also found that the speedups were smaller than the numbers that we got from the previous version of our system, where forward error correction was not used. We performed some initial profiling of the time spent for operation shipping and found that the overhead of FEC was not small, sometimes as high as 80% of the total elapsed time. Although FEC is useful in handling the side effects of timestamps, it does not justify such a large overhead. We plan to use two optimizations to reduce the overhead: (1) we could use FEC on only those applications that need it, using hints from the users, and (2) we could choose to use a smaller number of parity symbols (says, 16) and substantially reduce the amount of computation needed.

Even without the planned optimizations, our current result has already shown that operation shipping is useful. Our result also indicates another advantage of operation shipping. That is, the speed of update propagation is much less sensitive to the network condition. This can be seen from the elapsed-time–bandwidth curves for test T9, plotted in Figure 7, in which the curves for value shipping is steep and that for operation shipping is flat. (Curves for other tests show similar trends.)

Combining the results of these two subsections, we conclude that operation shipping can reduce network traffic very substantially, can accelerate update propagation substantially, and can make the elapsed time of update propagation much less dependent on the network condition.

		Data size	Speedup		
Test	Nature	(Kbytes)	9.6	28.8	64
T1	rp2gen	27.5	3.4	1.5	0.8
T2	rp2gen	76.3	7.7	3.3	1.7
T3	yacc	23.5	6.9	3.2	1.7
T4	c++ -c	26.0	4.9	2.5	1.4
T5	c++ -c	62.4	7.8	3.4	1.8
T6	c++ -c	265.3	16.4	6.4	3.0
T7	c++	23.0	5.0	2.7	1.6
T8	ar	69.3	11.7	5.4	2.9
T9	ar	363.1	26.3	10.2	5.1
T10	tar x	269.5	19.3	9.0	4.7
T11	make	69.9	7.6	3.3	1.7
T12	make	237.1	10.1	4.3	2.1
T13	tar c	60.0	15.1	6.3	3.4
T14	sgml2latex	41.8	7.1	2.8	1.4
T15	sgml2latex	270.0	17.8	6.5	3.1
T16	latex	93.4	10.2	4.1	2.2

Speedups for update propagation under three different network speeds: 9.6-Kbps (9.6), 28.8-Kbps (28.8), and 64-Kbps (64).

Figure 6: Speedups for update propagation

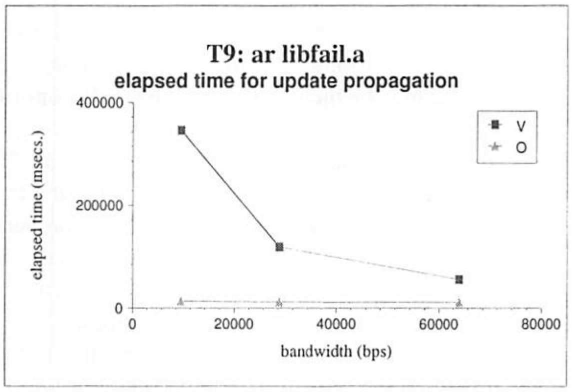


Figure 7: Elapsed time vs. bandwidth

5 Related work and alternative solutions

5.1 Related work

To the best of our knowledge, this is the first work that attempts to propagate file updates by operation. However, some ideas and techniques used in this work have been studied in previous research.

Uses in database. The idea of operation-based update propagation is not new to the database community [15], but we apply it in a new context: distributed file system. First of all, we need to log and ship operations at a level higher than the file system itself, because the low-level file-system operations are not appropriate for operation shipping. Therefore, we need cooperation between the applications (or meta-applications) and the file system. Also, the new context requires several new concepts: re-execution by surrogate, adjustment of meta-data, validation of re-execution, and handling of non-repeating side effects. Finally, our file system can attempt operation shipping more boldly, because it has a fall-back mechanism of value shipping.

Directory operations. Logging and shipping of directory operations have been implemented in Coda prior to this work [20, 19]. When a directory is updated on a Coda client (e.g., a new entry is inserted), instead of shipping the whole new directory to the server, the client ships only the update operation (e.g., the insertion operation). Directory operations are more like database operations, since they can be mapped directly to insertion, deletion, and modification of directory entries. In contrast, this work focuses on operation shipping of general user operations.

Repeatable re-execution. Several previous research projects have investigated the conditions for repeatable re-executions. Repeatable re-executions were desired for fault tolerance [1] or load balancing [2]. In the former case, a process P can be backed up by another process P_b. If P crashes, then P_b will repeat the execution of P since a recent checkpoint, and will thereafter assume the role of P. In the latter case, a process can migrate to another host to reduce the load imposed on the original host. In our work, repeatable re-executions are used to re-produce some file modifications that are identical to those produced by the original executions.

Re-execution for transactional guarantee. A previous Coda project has implemented a mechanism for re-execution of operations [9] [10]. It addresses the update conflicts that may be incurred in optimistically controlled replica. It proposes that a user can declare a portion of execution as an Isolation-Only Transaction (IOT). If an update conflict happens, Coda will re-execute the transaction. Our work is different in that we focus on performance improvement. Also, in our work, re-executions take place in a different host, whereas re-execution of IOTs take place in the same host. This implies that we must handle the case where re-execution does not produce the same results as the original execution.

5.2 Alternative solutions

There are other possible solutions to the problem that we are addressing. We are going to discuss four of them.

Delta shipping. The idea is to ship only the incremental difference, which is also called the *delta*, between different versions of a file. It has been proposed by many people and is currently being used as a general mechanism [24] or in specific systems including file systems [5], web proxies [12], file archives [11], and source-file repositories [22, 16].

It is possible to compute deltas not only for text files but also for binary files. We would like to mention the `rsync` algorithm [24] in particular. When shipping a file, the sending host suppresses the shipping of some blocks of data if they are found to be present on the receiving host already. It determines whether they are already present on the receiving host by using the checksum information supplied by the receiving host. The algorithm exploits a rolling checksum algorithm so that the blocks being matched can be started at any offset, not just multiples of block size.

Delta shipping has several limitations. First, a newly-created file has no previous version. Second, the effectiveness of delta shipping largely depends on how similar the two versions of a file are, and how those incremental differences are distributed in the file. In pathological case, a slightly changed file may need a huge delta. This could happen, for example, if there are some global substitutions of strings, or if the brightness or contrast of an image is changed. In general, we believe operation shipping can achieve a larger reduction of network traffic.

On the other hand, delta shipping does not involve re-execution of applications and pre-arrangement of surrogate clients, as operation shipping does. Therefore, it is simpler in terms of system administration. We believe delta shipping and operation shipping can complement each other in a distributed file system. In particular, delta shipping can be used when operation shipping has failed for some updates, and when the file system has resorted to use value shipping.

Data compression. Data compression reduces the size of a file by taking out the redundancy in the file. This technique can be used in a file system or a web proxy [12]. However, the reduction factors achieved by data compression may be smaller than that of operation shipping. We did a small performance study using a representative implementation: the `gzip` utility, which uses the Lempel-Ziv coding (LZ77). We ran gzip with the updated files of the 16 tests in Section 4, and listed the expected traffic volume and expected traffic reduction by compressing the files before shipping them. The results are shown in Figure 4 (the sixth and seventh column). The expected traffic reductions by data compression ranged from 2.7 to 8.1, substantially smaller than that achieved by operation shipping, which ranged from 12.0 to 245.7. We were not surprised by the results, since we know operation shipping exploits the semantic information of the user operation, whereas data compression operates only generically on the files. Like delta shipping, data compression can complement operation shipping, and be used when our file system has resorted to value shipping.

Logging keystrokes. A file system may log keystrokes and mouse clicks, ship them, and replay them on the surrogate. As such, it may be transparent to an application even if the application is interactive. However, we are pessimistic about this approach, because it is very difficult to make sure the logged keystrokes and mouse clicks will produce the identical outcome on the surrogate machine. Too many things can happen at run-time that could cause the keystrokes to produce different results.

Operation shipping without involving the file system. Can we use operation shipping without involving the file system? We can imagine that someone may design a meta-application that logs every command a user types, and, without involving the file system, remotely executes the same commands on a surrogate machine. We believe such a system would not work, for the following reasons. If the file system had no knowledge that the second execution was a re-execution, it would treat the files produced by the two execution as two distinct copies, and would force the client to fetch the surrogate copy. It might even think that there was an update/update conflict. Besides, it cannot ensure the correctness of the re-execution. We therefore believe that the file system plays a key role in useful and correct operation shipping,

6 Conclusion

Our experience with operation shipping, although it is limited only to the application-transparent case, is entirely favorable. We have implemented a prototype by extending the Coda File System, and have demonstrated that operation shipping can achieve substantial performance improvements.

Efficient update propagation is important for insulating users from the unpleasant consequences of low bandwidth networks. Indeed, without this capability, performance may be sufficiently degraded that users are tempted to forgo the transparency benefits of a distributed file system, and rely on explicit copying of local files instead. Our results suggest that operation shipping can play an important role in the design of future distributed file systems for bandwidth-challenged environments.

Acknowledgements

Matt Mathis gave us the idea of using forward error correction for handling the side effect of time stamps. Phil Karn allowed us to use his Reed-Solomon-Code package in our prototype. Maria Ebling and David Eckhardt provided major input for the high-level design of the system. Peter Braam insisted that operation logging must be as transparent as possible to non-interactive applications, and made us think more carefully about the issue. Many people, including John C. S. Lui, Peter T. S. Tam, K. Y. So, Eric Tilton, and Kip Walker, helped in improving the presentation of the paper. Finally, this work would not have been possible without the excellent support of the Coda community.

References

[1] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[2] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software–Practice and Experience*, 21(8):757–785, August 1991.

[3] The Coda Group. Coda File System. Available from http://coda.cs.cmu.edu.

[4] A. Houghton. *The Engineer's Error Coding Handbook*. Chapman & Hall, 1997.

[5] Airsoft Inc. Powerburst – The First Software Accelerator That More Than Doubles Remote Node Performance. Available from http://www.airsoft.com/comp.html, Cupertino, CA.

[6] P. Karn. Error Control Coding, a Seminar handout. Available from http://people.qualcomm.com/karn/dsp.html.

[7] J. J. Kistler. *Disconnected Operation in a Distributed File System*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1993.

[8] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[9] Q. Lu. *Improving Data Consistency for Mobile File Access Using Isolation-Only Transaction*. PhD thesis, Carnegie Mellon University, School of Computer Science, May 1996.

[10] Q. Lu and M. Satyanarayanan. Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions. In *Proceedings of the Fifth IEEE HotOS Topics Workshop*, Orcas Island, WA, May 1995.

[11] J. MacDonald. Versioned File Archiving, Compression, and Distribution. submitted for the Data Compression Conference, an earlier version is available from http://www.XCF.Berkeley.edu/~jmacd/xdelta.html, 1998.

[12] J.C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceeding of the ACM SIGCOMM'97*, 1997.

[13] L. B. Mummert. *Exploiting Weak Connectivity in a Distributed File System.* PhD thesis, Carnegie Mellon University, School of Computer Science, 1996.

[14] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, December 1995.

[15] K. Patersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operatiog Systems Principles*, Saint-Malo, France, October 1997.

[16] The FreeBSD Documentation Project. CVSup: in FreeBSD Handbook. Available from http://www.freebsd.org/handbook/cvsup.html.

[17] R. Rivest. The MD5 Message-Digest Algorithm, Internet RFC 1321. Available from http://theory.lcs.mit.edu/~rivest/publications.html, April 1992.

[18] M. Satyanarayanan, M. R. Ebling, J. Raiff, and P. J. Braam. *Coda File System User and System Administrators Manual.* School of Computer Science, Carnegie Mellon University, August 1997. version 1.1.

[19] M. Satyanarayanan, J. J. Kistler, P. Kumar, and H. Mashburn. On the Ubiquity of Logging in Distributed File Systems. In *Third IEEE Workship on Workstation Operation Systems*, Key Biscayne, FL, Apr 1992.

[20] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transaction on Computers*, 39(4), April 1990.

[21] B. Schneier. *Applied Cryptography.* John Wiley & Sons, Inc., second edition, 1996.

[22] Cyclic Software. Concurrent Versions System (CVS). Available from http://www.cyclic.com/.

[23] W. R. Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley, 1992.

[24] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, The Australian National University, Available from http://samba.anu.edu.au/rsync/, June 1996.

Extending File Systems Using Stackable Templates

Erez Zadok, Ion Badulescu, and Alex Shender
Computer Science Department, Columbia University
{ezk,ion,alex}@cs.columbia.edu

Abstract

Extending file system functionality is not a new idea, but a desirable one nonetheless[6, 14, 18]. In the several years since stackable file systems were first proposed, only a handful are in use[12, 19]. Impediments to writing new file systems include the complexity of operating systems, the difficulty of writing kernel-based code, the lack of a true stackable vnode interface[14], and the challenges of porting one file system to another operating system.

We advocate writing new stackable file systems as kernel modules. As a starting point, we propose a portable, stackable template file system we call Wrapfs (wrapper file system). Wrapfs is a canonical, minimal stackable file system that can be used as a pattern across a wide range of operating systems and file systems. Given Wrapfs, developers can add or modify only that which is necessary to achieve the desired functionality. Wrapfs takes care of the rest, and frees developers from the details of operating systems. Wrapfs templates exist for several common operating systems (Solaris, Linux, and FreeBSD), thus alleviating portability concerns. Wrapfs can be ported to any operating system with a vnode interface that provides a private data pointer for each data structure used in the interface. The overhead imposed by Wrapfs is only 5–7%.

This paper describes the design and implementation of Wrapfs, explores portability issues, and shows how the implementation was achieved without changing client file systems or operating systems. We discuss several examples of file systems written using Wrapfs.

1 Introduction

Adding functionality to existing file systems in an easy manner has always been desirable. Several ideas have been proposed and some prototypes implemented[6, 14, 18]. None of the proposals for a new extensible file system interface has made it to commonly used Unix operating systems. The main reasons are the significant changes that overhauling the file system interface would require, and the impact it would have on performance.

Kernel-resident native file systems are those that interact directly with lower level media such as disks[9] and networks[11, 16]. Writing such file systems is difficult because it requires deep understanding of specific operating system internals, especially the interaction with device drivers and the virtual memory system. Once such a file system is written, porting it to another operating system is just as difficult as the initial implementation, because specifics of different operating systems vary significantly.

Others have resorted to writing file systems at the user level. These file systems work similarly to the Amd automounter[13] and are based on an NFS server. While it is easier to develop user-level file servers, they suffer from poor performance due to the high number of context switches they incur. This limits the usefulness of such file systems. Later works, such as Autofs[3], attempt to solve this problem by moving critical parts of the automounter into the kernel.

We propose a compromise solution to these problems: writing kernel resident file systems that use existing native file systems, exposing to the user a vnode interface that is similar even across different operating systems. Doing so results in performance similar to that of kernel-resident systems, with development effort on par with user level file systems. Specifically, we provide a template *Wrapper File System* called Wrapfs. Wrapfs can *wrap* (mount) itself on top of one or more existing directories, and act as an intermediary between the user accessing the mount point and the lower level file system it is mounted on. Wrapfs can transparently change the behavior of the file system, while keeping the underlying media unaware of the upper-level changes. The Wrapfs template takes care of many file system internals and operating system bookkeeping, and it provides the developer with simple hooks to manipulate the data and attributes of the lower file system's objects.

1.1 The Stackable Vnode Interface

Wrapfs is implemented as a stackable vnode interface. A *Virtual Node* or *vnode* is a data structure used within Unix-based operating systems to represent an open file, directory, or other entities that can appear in the file system namespace. A vnode does not expose what type of physical file system it implements. The *vnode interface* allows higher level operating system modules to perform operations on vnodes uniformly. The *virtual file system* (VFS) contains the common file system code of the vnode interface.

One improvement to the vnode concept is *vnode stacking*[6, 14, 18], a technique for modularizing file system functions by allowing one vnode interface implementation to call another. Before stacking existed, there was only one vnode interface implementation; higher level operating system code called the vnode interface which in turn called code for a specific file system. With vnode stacking, several implementations may exist and may call each other in sequence: the code for a certain operation at stack level N typically calls the corresponding operation at level $N-1$, and so on.

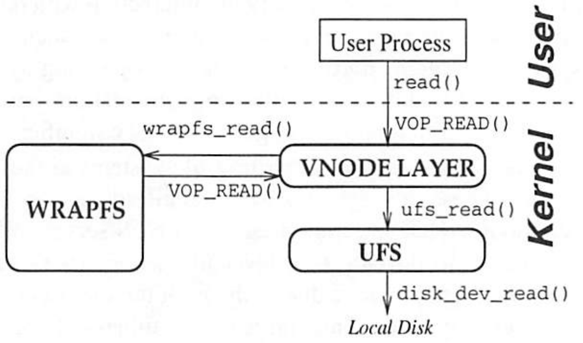


Figure 1: A Vnode Stackable File System

Figure 1 shows the structure for a simple, single-level stackable wrapper file system. System calls are translated into VFS calls, which in turn invoke their Wrapfs equivalents. Wrapfs again invokes generic VFS operations, and the latter call their respective *lower level* file system operations. Wrapfs calls the lower level file system without knowing who or what type it is.

The rest of this paper is organized as follows. Section 2 discusses the design of Wrapfs. Section 3 details Wrapfs's implementation, and issues relating to its portability to various platforms. Section 4 describes four example file systems written using Wrapfs; Section 5 evaluates their performance and portability. We survey related works in Section 6 and conclude in Section 7.

2 Design

Our main design considerations for Wrapfs were:

1. What developers want to change in a file system.
2. What API should Wrapfs offer these users that would free them from operating system internals.
3. How to allow advanced users the flexibility to control and manipulate all aspects of the file system.
4. Interaction of caches among different layers.
5. What user level mounting-related issues are there.
6. Performance overhead of Wrapfs.

The first five points are discussed below. Performance is addressed in detail in Section 5.

2.1 What to Change in a File System

As shown in Figure 1, Wrapfs is independent of the host system's vnode interface. Since most UNIX-like systems (including those that currently support Wrapfs) have static vnode interfaces, this means that Wrapfs cannot introduce fundamentally new vnode operations.[1] (Limited new functionality can be added using new `ioctl(2)` calls.) Our stackable file system architecture can, however, manipulate data, names, and attributes of files. We let Wrapfs users manipulate any of these.

The most obvious manipulation Wrapfs users want to do is of file data; that is useful for many applications, for example in encryption file systems. The next most likely item to manipulate is the file name. For example, an encryption file system can encrypt file names as well as data. A file system that translates between Unix and MS-DOS style file names can uniquely map long mixed-case Unix file names to 8.3-format upper-case names.

Finally, there are file attributes that users might want to change. For example, a file system can ensure that all files within it are world readable, regardless of the individual umask used by the users creating them. This is useful in directories where all files must be world-readable (such as html files served by most Web servers). Another file system might prevent anyone from setting the uid bit on executables, as a simple form of intrusion avoidance.

The aforementioned list of examples is not exhaustive, but only a hint of what can be accomplished with level of flexibility that Wrapfs offers. Wrapfs's developer API is summarized in Table 1 and is described below.

2.1.1 File Data API

The system call interface offers two methods for reading and writing file data. The first is by using the `read` and `write` system calls. The second is via the MMAP interface. The former allows users to manipulate arbitrary amounts of file data. In contrast, the MMAP interface operates on a file in units of the native page size. To accommodate the MMAP interface, we decided to require file system developers using Wrapfs to also manipulate file data on whole pages. Another reason for manipulating only whole pages was that some file data changes may require it. Some encryption algorithms work on fixed size data blocks and bytes within the block depend on preceding bytes.

[1]The UCLA stackable file system replaced the static UNIX vnode interface with a dynamic interface that allowed file system developers to introduce new operations[6].

Call	Input Argument	Output Argument
encode_data	buffer from user space	encoded (same size) buffer to be written
decode_data	buffer read from lower level file system	decoded (same size) buffer to pass to user space
encode_filename	file name passed from user system call	encoded (and allocated) file name of any length to use in lower level file system
decode_filename	file name read from the lower level file system	decoded (and allocated) file name of any length to pass back to a user process
other	Inspect or modify file attributes in vnode functions, right before or after calling lower level file system	

Table 1: Wrapfs Developer API

All vnode calls that write file data call a function `encode_data` before writing the data to the lower level file system. Similarly, all vnode calls that read file data call a function `decode_data` after reading the data from the lower level file system. These two functions take two buffers of the same size: one as input, and another as output. The size of the buffer can be defined by the Wrapfs developer, but it must be an even multiple of the system's page size, to simplify handling of MMAP functions. Wrapfs passes other auxiliary data to the encode and decode functions, including the file's attributes and the user's credentials. These are useful when determining the proper action to take. The encode and decode functions return the number of bytes manipulated, or a negative error code.

All vnode functions that manipulate file data, including the MMAP ones, call either the encode or decode functions at the right place. Wrapfs developers who want to modify file data need not worry about the interaction between the MMAP, read, and write functions, about file or page locks, reference counts, caches, status flags, and other bookkeeping details; developers need only to fill in the encode and decode functions appropriately.

2.1.2 File Names API

Wrapfs provides two file name manipulating functions: `encode_filename` and `decode_filename`. They take in a single file name component, and ask the Wrapfs developer to fill in a new encoded or decoded file name of any length. The two functions return the number of bytes in the newly created string, or a negative error code. Wrapfs also passes to these functions the file's vnode and the user's credentials, allowing the function to use them to determine how to encode or decode the file name. Wrapfs imposes only one restriction on these file name manipulating functions. They must not return new file names that contain characters illegal in Unix file names, such as a null or a "/".

The user of Wrapfs who wishes to manipulate file names need not worry about which vnode functions use file names, or how directory reading (`readdir`) is being accomplished. The file system developer need only fill in the file name encoding and decoding functions. Wrapfs takes care of all other operating system internals.

2.1.3 File Attributes

For the first prototype of Wrapfs, we decided not to force a specific API call for accessing or modifying file attributes. There are only one or two places in Wrapfs where attributes are handled, but these places are called often (i.e., lookup). We felt that forcing an API call might hurt performance too much. Instead, we let developers inspect or modify file attributes directly in Wrapfs's source.

2.2 User Level Issues

There are two important issues relating to the extension of the Wrapfs API to user-level: mount points and ioctl calls.

Wrapfs can be mounted as a regular mount or an overlay mount. The choice of mount style is left to the Wrapfs developer. Figure 2 shows an original file system and the two types of mounts, and draws the boundary between Wrapfs and the original file system after a successful mount.

In a regular mount, Wrapfs receives two pathnames: one for the mount point (`/mnt`), and one for the directory to stack on (the mounted directory `/usr`). After executing, for example, `mount -t wrapfs /mnt /usr`, there are two ways to access the mounted-on file system. Access via the mounted-on directory (`/usr/ucb`) yields the lower level files without going through Wrapfs. Access via the mount point (`/mnt/ucb`), however, goes through Wrapfs first. This mount style exposes the mounted directory to user processes; it is useful for debugging purposes and for applications (e.g., backups) that do not need the functionality Wrapfs implements. For example, in an encryption file system, a backup utility can backup files faster and safer if it uses the lower file system's files (ciphertext), rather than the ones through the mount point (cleartext).

In an overlay mount, accomplished using `mount -t wrapfs -O /usr`, Wrapfs is mounted directly on top of `/usr`. Access to files such as `/usr/ucb` go though Wrapfs. There is no way to get to the original file system's files under `/usr` without passing through Wrapfs first. This mount style has the advantage of hiding the lower level file system from user processes, but may make backups and debugging harder.

The second important user-level issue relates to the `ioctl(2)` system call. Ioctls are often used to extend file

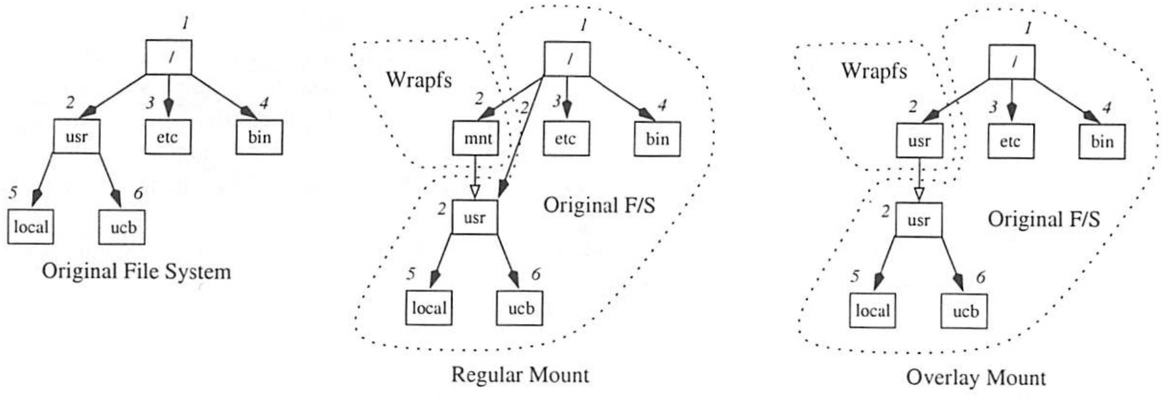


Figure 2: Wrapfs Mount Styles

system functionality. Wrapfs allows its user to define new ioctl codes and implement their associated actions. Two ioctls are already defined: one to set a debugging level and one to query it. Wrapfs comes with lots of debugging traces that can be turned on or off at run time by a root user. File systems can implement other ioctls. An encryption file system, for example, can use an ioctl to set encryption keys.

2.3 Interaction Between Caches

When Wrapfs is used on top of a disk-based file system, both layers cache their pages. Cache incoherency could result if pages at different layers are modified independently. A mechanism for keeping caches synchronized through a centralized cache manager was proposed by Heidemann[5]. Unfortunately, that solution involved modifying the rest of the operating system and other file systems.

Wrapfs performs its own caching, and does not explicitly touch the caches of lower layers. This keeps Wrapfs simpler and more independent of other file systems. Also, since pages can be served off of their respective layers, performance is improved. We decided that the higher a layer is, the more authoritative it is: when writing to disk, cached pages for the same file in Wrapfs overwrite their UFS counterparts. This policy correlates with the most common case of cache access, through the uppermost layer. Finally, note that a user process can access cached pages of a lower level file system only if it was mounted as a regular mount (Figure 2). If Wrapfs is overlay mounted, user processes could not access lower level files directly, and cache incoherency for those files is less likely to occur.

3 Implementation

This section details the more difficult parts of the implementation and unexpected problems we encountered. Our first implementation concentrated on the Solaris 2.5.1 operating system because Solaris has a standard vnode interface and we had access to kernel sources. Our next two implementations were for the Linux 2.0 and the FreeBSD 3.0 operating systems. We chose these two because they are popular, are sufficiently different, and they also come with kernel sources. In addition, all three platforms support loadable kernel modules, which made debugging easier. Together, the platforms we chose cover a large portion of the Unix market.

The discussion in the rest of this section concentrates mostly on Solaris, unless otherwise noted. In Section 3.5 we discuss the differences in implementation between Linux and Solaris. Section 3.6 discusses the differences for the FreeBSD port.

3.1 Stacking

Wrapfs was initially similar to the Solaris loopback file system (lofs)[19]. Lofs passes all Vnode/VFS operations to the lower layer, but it only stacks on directory vnodes. Wrapfs stacks on every vnode, and makes identical copies of data blocks, pages, and file names in its own layer, so they can be changed independently of the lower level file system. Wrapfs does not explicitly manipulate objects in other layers. It appears to the upper VFS as a lower-level file system; concurrently, Wrapfs appears to lower-level file systems as an upper-layer. This allows us to stack multiple instances of Wrapfs on top of each other.

The key point that enables stacking is that each of the major data structures used in the file system (`struct vnode` and `struct vfs`) contain a field into which we can store file system specific data. Wrapfs uses that private field to store several pieces of information, especially a pointer to the corresponding lower level file system's vnode and VFS. When a vnode operation in Wrapfs is called, it finds the lower level's vnode from the current vnode, and repeats the same operation on the lower level vnode.

3.2 Paged Reading and Writing

We perform reading and writing on whole blocks of size matching the native page size. Whenever a read for a range of bytes is requested, we compute the extended range of bytes up to the next page boundary, and apply the operation to the lower file system using the extended range. Upon successful completion, the exact number of bytes requested are returned to the caller of the vnode operation.

Writing a range of bytes is more complicated than reading. Within one page, bytes may depend on previous bytes (e.g., encryption), so we have to read and decode parts of pages before writing other parts of them.

Throughout the rest of this section we will refer to the upper (wrapping) vnode as V, and to the lower (wrapped) vnode as V'; P and P' refer to memory mapped pages at these two levels, respectively. The example[2] depicted in Figure 3 shows what happens when a process asks to write bytes of an existing file from byte 9000 until byte 25000. Let us assume that the file in question has a total of 4 pages (32768) worth of bytes in it.

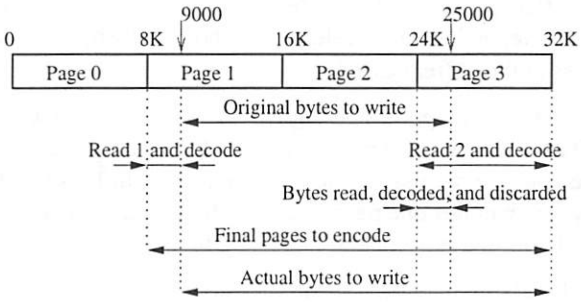


Figure 3: Writing Bytes in Wrapfs

1. Compute the extended page boundary for the write range as 8192–32767 and allocate three empty pages. (Page 0 of V is untouched.)
2. Read bytes 8192–8999 (page 1) from V', decode them, and place them in the first allocated page. We do not need to read or decode the bytes from 9000 onwards in page 1 because they will be overwritten by the data we wish to write anyway.
3. Skip intermediate data pages that will be overwritten by the overall write operation (page 2).
4. Read bytes 24576–32767 (page 3) from V', decode them, and place them in the third allocated page. This time we read and decode the whole page because we need the last 32767−25000=7767 bytes and these bytes depend on the first 8192−7767=426 bytes of that page.
5. Copy the bytes that were passed to us into the appropriate offset in the three allocated pages.
6. Finally, we encode the three data pages and call the write operation on V' for the same starting offset (9000). This time we write the bytes all the way to the last byte of the last page processed (byte 32767), to ensure validity of the data past file offset 25000.

[2]The example is simplified because it does not take into account sparse files, and appending to files.

3.2.1 Appending to Files

When files are opened for appending only, the VFS does not provide the vnode `write` function the real size of the file and where writing begins. If the size of the file before an append is not an exact multiple of the page size, data corruption may occur, since we will not begin a new encoding sequence on a page boundary.

We solve this problem by detecting when a file is opened with an append flag on, turn off that flag before the open operation is passed on to V', and replace it with flags that indicate to V' that the file was opened for normal reading and writing. We save the initial flags of the opened file, so that other operations on V could tell that the file was originally opened for appending. Whenever we write bytes to a file that was opened in append-only mode, we first find its size, and add that to the file offsets of the write request. In essence we convert append requests to regular write requests starting at the end of the file.

3.3 File Names and Directory Reading

Readdir is implemented in the kernel as a restartable function. A user process calls the readdir C library call, which is translated into repeated calls to the `getdents(2)` system call, passing it a buffer of a given size. The kernel fills the buffer with as many directory entries as will fit in the caller's buffer. If the directory was not read completely, the kernel sets a special EOF flag to false. As long as the flag is false, the C library function calls `getdents(2)` again.

The important issue with respect to directory reading is how to continue reading the directory from the offset where the previous read finished. This is accomplished by recording the last position and ensuring that it is returned to us upon the next invocation. We implemented `readdir` as follows:

1. A readdir vnode operation is called on V for N bytes worth of directory data.
2. Call the same vnode operation on V' and read back N bytes.
3. Create a new temporary buffer of a size that is as large as N.
4. Loop over the bytes read from V', breaking them into individual records representing one directory entry at a time (`struct dirent`). For each such, we call `decode_filename` to find the original file name. We construct a new directory entry record containing the decoded file name and add the new record to the allocated temporary buffer.
5. We record the offset to read from on the next call to `readdir`; this is the position past the last file name we just read and decoded. This offset is stored in one of the fields of the `struct uio` (representing data movement between user and kernel space) that is returned to the caller. A new structure is passed to

us upon the next invocation of readdir with the offset field untouched. This is how we are able to restart the call from the correct offset.

6. The temporary buffer is returned to the caller of the vnode operation. If there is more data to read from V', then we set the EOF flag to false before returning from this function.

The caller of `readdir` asks to read at most N bytes. When we decode or encode file names, the result can be a longer or shorter file name. We ensure that we fill in the user buffer with no more `struct dirent` entries than could fit (but fewer is acceptable). Regardless of how many directory entries were read and processed, we set the file offset of the directory being read such that the next invocation of the `readdir` vnode operation will resume reading file names from exactly where it left off the last time.

3.4 Memory Mapping

To support MMAP operations and execute binaries we implemented memory-mapping vnode functions. As per Section 2.3, Wrapfs maintains its own cached decoded pages, while the lower file system keeps cached encoded pages.

When a page fault occurs, the kernel calls the vnode operation `getpage`. This function retrieves one or more pages from a file. For simplicity, we implemented it as repeatedly calling a function that retrieves a single page—`getapage`. We implemented `getapage` as follows:

1. Check if the page is cached; if so, return it.
2. If the page is not cached, create a new page P.
3. Find V' from V and call the `getpage` operation on V', making sure it would return only one page P'.
4. Copy the (encoded) data from P' to P.
5. Map P into kernel virtual memory and decode the bytes by calling `wrapfs_decode`.
6. Unmap P from kernel VM, insert it into V's cache, and return it.

The implementation of `putpage` was similar to `getpage`. In practice we also had to carefully handle two additional details, to avoid deadlocks and data corruption. First, pages contain several types of locks, and these locks must be held and released in the right order and at the right time. Secondly, the MMU keeps mode bits indicating status of pages in hardware, especially the referenced and modified bits. We had to update and synchronize the hardware version of these bits with their software version kept in the pages' flags. For a file system to have to know and handle all of these low-level details blurs the distinction between the file system and the VM system.

3.5 Linux

When we began the Solaris work we referred to the implementation of other file systems such as lofs. Linux 2.0 did not have one as part of standard distributions, but we were able to locate and use a prototype[3]. Also, the Linux Vnode/VFS interface contains a different set of functions and data structures than Solaris, but it operates similarly.

In Linux, much of the common file system code was extracted and moved to a generic (higher) level. Many generic file system functions exist that can be used by default if the file system does not define its own version. This leaves the file system developer to deal with only the core issues of the file system. For example, Solaris User I/O (`uio`) structures contain various fields that must be updated carefully and consistently. Linux simplifies data movement by passing I/O related vnode functions a simple allocated (`char *`) buffer and an integer describing how many bytes to process in the buffer passed.

Memory-mapped operations are also easier in Linux. The vnode interface in Solaris includes functions that must be able to manipulate one or more pages. In Linux, a file system handles one page at a time, leaving page clustering and multiple-page operations to the higher VFS.

Directory reading was simpler in Linux. In Solaris, we read a number of raw bytes from the lower level file system, and parse them into chunks of `sizeof(struct dirent)`, set the proper fields in this structure, and append the file name bytes to the end of the structure (out of band). In Linux, we provide the kernel with a callback function for iterating over directory entries. This function is called by higher level code and ask us to simply process one file name at a time.

There were only two caveats to the portability of the Linux code. First, Linux keeps a list of exported kernel symbols (in `kernel/ksyms.c`) available to loadable modules. To make Wrapfs a loadable module, we had to export additional symbols to the rest of the kernel, for functions mostly related to memory mapping. Second, most of the structures used in the file system (`inode`, `super_block`, and `file`) include a private field into which stacking specific data could be placed. We had to add a private field to only one structure that was missing it, the `vm_area_struct`, which represents custom per-process virtual memory manager page-fault handlers. Since Wrapfs is the first fully stackable file system for Linux, we feel that these changes are small and acceptable, given that more stackable file systems are likely to be developed.[4]

[3]http://www.kvack.org/~blah/lofs/

[4]We submitted our small changes and expect that they will be included in a future version of Linux.

3.6 FreeBSD

FreeBSD 3.0 is based on BSD-4.4Lite. We chose it as the third port because it represents another major section of Unix operating systems. FreeBSD's vnode interface is similar to Solaris's and the port was straightforward. FreeBSD's version of the loopback file system is called *nullfs*[12], a template for writing stackable file systems. Unfortunately, ever since the merging of the VM and Buffer Cache in FreeBSD 3.0, stackable file systems stopped working because of the inability of the VFS to correctly map data pages of stackable file systems to their on-disk locations. We worked around two deficiencies in nullfs. First, writing large files resulted in some data pages getting zero-filled on disk; this forced us to perform all writes synchronously. Second, memory mapping through nullfs paniced the kernel, so we implemented MMAP functions ourselves. We implemented `getpages` and `putpages` using `read` and `write`, respectively, because calling the lower-level's page functions resulted in a UFS pager error.

4 Examples

This section details the design and implementation of four sample file systems we wrote based on Wrapfs. The examples range from simple to complex:

1. **Snoopfs**: detects and warns of attempted access to users' files by other non-root users.
2. **Lb2fs**: is a read-only file system that trivially balances the load between two replicas of the same file system.
3. **Usenetfs**: breaks large flat article directories (often found in very active news spools) into deeper directory hierarchies, improving file access times.
4. **Cryptfs**: is an encryption file system.

These examples are experimental and intended to illustrate the kinds of file systems that can be written using Wrapfs. We do not consider them to be complete solutions. Whenever possible, we illustrate potential enhancements to our examples. We hope to convince readers of the flexibility and simplicity of writing new file systems using Wrapfs.

4.1 Snoopfs

Users' home directory files are often considered private and personal. Normally, these files are read by their owner or by the root user (e.g., during backups). Other sanctioned file access includes files shared via a common Unix group. Any other access attempt may be considered a break-in attempt. For example, a manager might want to know if a subordinate tried to `cd` to the manager's `~/private` directory; an instructor might wish to be informed when anyone tries to read files containing homework solutions.

The one place in a file system where files are initially searched is the vnode `lookup` routine. To detect access problems, we first perform the lookup on the lower file system, and then check the resulting status. If the status was one of the error codes "permission denied" or "file not found," we know that someone was trying to read a file they do not have access to, or they were trying to guess file names. If we detect one of these two error codes, we also check if the current process belongs to the super-user or the file's owner by inspecting user credentials. If it was a root user or the owner, we do nothing. Otherwise we print a warning using the in-kernel log facility. The warning contains the file name to which access was denied and the user ID of the process that tried to access it.

We completed the implementation of Snoopfs in less than one hour (on all three platforms). The total number of lines of C code added to Wrapfs was less than 10.

Snoopfs can serve as a prototype for a more elaborate intrusion detection file system. Such a file system can prohibit or limit the creation or execution of setuid/setgid programs; it can also disallow overwriting certain executables that rarely change (such as `/bin/login`) to prevent attackers from replacing them with trojans.

4.2 Lb2fs

Lb2fs is a trivial file system that multiplexes file access between two identical replicas of a file system, thus balancing the load between them. To avoid concurrency and consistency problems associated with writable replicas, Lb2fs is a read-only file system: vnode operations that can modify the state of the lower file system are disallowed. The implementation was simple; operations such as `write`, `mkdir`, `unlink`, and `symlink` just return the error code "read-only file system." We made a simplifying assumption that the two replicas provide service of identical quality, and that the two remote servers are always available, thus avoiding fail-over and reliability issues.

The one place where new vnodes are created is in the `lookup` function. It takes a directory vnode and a pathname and it returns a new vnode for the file represented by the pathname within that directory. Directory vnodes in Lb2fs store not one, but two vnodes of the lower level file systems—one for each replica; this facilitates load-balancing lookups in directories. Only non-directories stack on top of one vnode, the one randomly picked. Lb2fs's lookup was implemented as follows:

1. An operation `lookup` is called on directory vnode DV and file name X.
2. Get from DV the two lower vnodes DV_1' and DV_2'.
3. Pick one of the two lower vnodes at random, and repeat the lookup operation on it using X.
4. If the lookup operation for X was successful, then check the resulting vnode. If the resulting vnode was not a directory vnode, store it in the private data of DV and return.

5. If the resulting vnode was a directory vnode, then repeat the lookup operation on the *other* lower vnode; store the two resulting directory vnodes (representing X on the two replicas) in the private data of DV.

The implications of this design and implementation are twofold. First, once a vnode is created, all file operations using it go to the file server that was randomly picked for it. A lookup followed by an open, read, and close of a file, will all use the same file server. In other words, the granularity of our load balancing is on a per-file basis.

Second, since lookups happen on directory vnodes, we keep the two lower directory vnodes, one per replica. This is so we can randomly pick one of them to lookup a file. This design implies that every open directory vnode is opened on both replicas, and only file vnodes are truly randomly picked and load-balanced. The overall number of lookups performed by Lb2fs is twice for directory vnodes and only once for file vnodes. Since the average number of files on a file system is much larger than the number of directories, and directory names and vnodes are cached by the VFS, we expect the performance impact of this design to be small.

In less than one day we designed, implemented, tested, and ported Lb2fs. Many possible extensions to Lb2fs exist. It can be extended to handle three or a variable number of replicas. Several additional load-balancing algorithms can be implemented: round-robin, LRU, the most responsive/available replica first, etc. A test for downed servers can be included so that the load-balancing algorithm can avoid using servers that recently returned an I/O error or timed out (fail-over). Servers that were down can be added once again to the available pool after another timeout period.

4.3 Usenetfs

One cause of high loads on news servers in recent years has been the need to process many articles in very large flat directories representing newsgroups such as *control.cancel* and *misc.jobs.offered*. Significant resources are spent on processing articles in these few newsgroups. Most Unix directories are organized as a linear unsorted sequence of entries. Large newsgroups can have hundreds of thousands of articles in one directory, resulting in delays processing any single article.

When the operating system wants to lookup an entry in a directory with N entries, it may have to search all N entries to find the file in question. Table 2 shows the frequency of all file system operations that use a pathname on our news spool over a period of 24 hours.

It shows that the bulk of all operations are for looking up files, so these should run very fast regardless of the directory size. Operations that usually run synchronously (`unlink` and `create`) account for about 10% of news

Operation	**Frequency**	**% Total**
Lookup	7068838	88.41
Unlink	432269	5.41
Create	345647	4.32
Readdir	38371	0.48
All other	110473	1.38
Total	7995598	100.00

Table 2: Frequency of File System Operations on a News Spool

spool activity and should also perform well on large newsgroups.

Usenetfs is a file system that rearranges the directory structure from being flat to one with small directories containing fewer articles. By breaking the structure into smaller directories, it improves the performance of looking up, creating, or deleting files, since these operations occur on smaller directories. The following sections summarize the design and implementation of Usenetfs. More detailed information is available in a separate report[23].

4.3.1 Design of Usenetfs

We had three design goals for Usenetfs. First, Usenetfs should not require changing existing news servers, operating systems, or file systems. Second, it should improve performance of these large directories enough to justify its overhead and complexity. Third, it should selectively manage large directories with little penalty to smaller ones.

The main idea for improving performance for large flat directories is to break them into smaller ones. Since article names are composed of sequential numbers, we take advantage of that. We create a hierarchy consisting of one thousand directories as depicted in Figure 4. We distribute articles across 1000 directories named 000

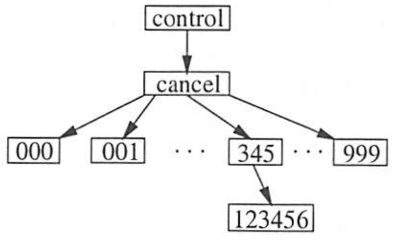


Figure 4: A Usenetfs Managed Newsgroup

through 999. Since article numbers are sequential, we maximize the distribution by computing the final directory into which the article will go based on three of the four least significant digits. For example, the article named `control/cancel/123456` is placed into the directory `control/cancel/345/`. We picked the directory based on the second, third, and fourth digits of the article number to allow for some amount of clustering. By not using the least significant digit we cluster 10 consecutive articles together: the articles 123450–123459 are placed in the same directory. This increases the chance of kernel cache hits due to the likelihood of sequential access of these articles. In general, every article numbered X..XYYYZ is placed in a directory named YYY. For reading a whole directory (`readdir`), we iterate over the subdirectories from 000 to 999, and return the entries within.

Usenetfs needs to determine if a directory is managed or not. We co-opted a seldom used mode bit for directories, the setuid bit, to flag a directory as managed by Usenetfs. Using this bit lets news administrators control which directories are managed, using a simple `chmod` command.

The last issue was how to convert an unmanaged directory to be managed by Usenetfs: creating some of the 000–999 subdirectories and moving existing articles to their designated locations. Experimentally, we found that the number of truly large newsgroups is small, and that they rarely shrunk. Given that, and for simplicity, we made the process of turning directory management on/off an off-line process triggered by the news administrator with a provided script.

4.3.2 Implementation of Usenetfs

Usenetfs is the first non-trivial file system we designed and implemented using Wrapfs. By "non-trivial" we mean that it took us more than a few hours to achieve a working prototype from the Wrapfs template. It took us one day to write the first implementation, and several more days to test it and alternate restructuring algorithms (discussed elsewhere[23]).

We accomplished most of the work in the functions `encode_filename` and `decode_filename`. They check the setuid bit of the directory to see if it is managed by Usenetfs; if so, they convert the filename to its managed representation and back.

4.4 Cryptfs

Cryptfs is the most involved file system we designed and implemented based on Wrapfs. This section summarizes its design and implementation. More detailed information is available elsewhere[24].

We used the Blowfish[17] encryption algorithm—a 64 bit block cipher designed to be fast, compact, and simple. Blowfish is suitable in applications where the keys seldom change such as in automatic file decryptors. It can use variable length keys as long as 448 bits. We used 128 bit keys.

We picked the Cipher Block Chaining (CBC) encryption mode because it allows us to encrypt byte sequences of any length—suitable for encrypting file names. We decided to use CBC only within each encrypted block. This way ciphertext blocks (of 4–8KB) do not depend on previous ones, allowing us to decrypt each block independently. Moreover, since Wrapfs lets us manipulate file data in units of page size, encrypting them promised to be simple.

To provide stronger security, we encrypt file names as well. We do not encrypt "." and ".." to keep the lower level Unix file system intact. Furthermore, since encrypting file names may result in characters that are illegal in file names (nulls and "/"), we uuencode the resulting encrypted strings. This eliminates unwanted characters and guarantees that all file names consist of printable valid characters.

4.4.1 Key Management

Only the root user is allowed to mount an instance of Cryptfs, but can not automatically encrypt or decrypt files. To thwart an attacker who gains access to a user's account or to root privileges, Cryptfs maintains keys in an in-memory data structure that associates keys not with UIDs alone but with the combination of UID and session ID. To acquire or change a user's key, attackers would not only have to break into an account, but also arrange for their processes to have the same session ID as the process that originally received the user's passphrase. This is a more difficult attack, requiring session and terminal hijacking or kernel-memory manipulations.

Using session IDs to further restrict key access does not burden users during authentication. Login shells and daemons use `setsid(2)` to set their session ID and detach from the controlling terminal. Forked processes inherit the session ID from their parent. Users would normally have to authorize themselves only once in a shell. From this shell they could run most other programs that would work transparently and safely with the same encryption key.

We designed a user tool that prompts users for passphrases that are at least 16 characters long. The tool hashes the passphrases using MD5 and passes them to Cryptfs using a special `ioctl(2)`. The tool can also instruct Cryptfs to delete or reset keys.

Our design decouples key possession from file ownership. For example, a group of users who wish to edit a single file would normally do so by having the file group-owned by one Unix group and add each user to that group. Unix systems often limit the number of groups a user can be a member of to 8 or 16. Worse, there are often many subsets of users who are all members of one group and wish to share certain files, but are unable to guarantee the security of their shared files because there are other users who are members of the same group; e.g., many sites put all of their staff members in a group called "staff," students in the "student" group, guests in another, and so on. With our design, users can further restrict access to shared files only to those users who were given the decryption key.

One disadvantage of this design is reduced scalability with respect to the number of files being encrypted and shared. Users who have many files encrypted with different keys have to switch their effective key before attempting to access files that were encrypted with a different one. We do not perceive this to be a serious problem for two reasons. First, the amount of Unix file sharing of restricted files is limited. Most shared files are generally world-readable and thus do not require encryption. Second, with the proliferation of windowing systems, users can associate different keys with different windows.

Cryptfs uses one Initialization Vector (IV) per mount, used to jump-start a sequence of encryption. If not specified, a predefined IV is used. A superuser mounting Cryptfs can choose a different IV, but that will make all previously encrypted files undecipherable with the new IV. Files that use the same IV and key produce identical ciphertext blocks that are subject to analysis of identical blocks. CFS[2] is a user level NFS-based encryption file system. By default, CFS uses a fixed IV, and we also felt that using a fixed one produces sufficiently strong security.

One possible extension to Cryptfs might be to use different IVs for different files, based on the file's inode number and perhaps in combination with the page number. Other more obvious extensions to Cryptfs include the use of different encryption algorithms, perhaps different ones per user, directory, or file.

5 Performance

When evaluating the performance of the file systems we built, we concentrated on Wrapfs and the more complex file systems derived from Wrapfs: Cryptfs and Usenetfs. Since our file systems are based on several others, our measurements were aimed at identifying the overhead that each layer adds. The main goal was to prove that the overhead imposed by stacking is acceptably small and comparable to other stacking work[6, 18].

5.1 Wrapfs

We include comparisons to a native disk-based file system because disk hardware performance can be a significant factor. This number is the base to which other file systems compare to. We include figures for Wrapfs (our full-fledged stackable file system) and for lofs (the low-overhead simpler one), to be used as a base for evaluating the cost of stacking. When using lofs or Wrapfs, we mounted them over a local disk based file system.

To test Wrapfs, we used as our performance measure a full build of Am-utils[22], a new version of the Berkeley Amd automounter. The test auto-configures the package and then builds it. Only the sources for Am-utils and the binaries they create used the test file system; compiler tools were left outside. The configuration runs close to seven hundred small tests, many of which are small compilations and executions. The build phase compiles about 50,000 lines of C code in several dozen files and links eight binaries. The procedure contains both CPU and I/O bound operations as well as a variety of file system operations.

For each file system measured, we ran 12 successive builds on a quiet system, measured the elapsed times of each run, removed the first measure (cold cache) and averaged the remaining 11 measures. The results are summarized in Table 3.[5] The standard deviation for the results reported in this section did not exceed 0.8% of the mean. Finally, there is no native lofs for FreeBSD, and the nullfs available is not fully functional (see Section 3.6).

File System	**SPARC 5**		**Intel P5/90**		
	Solaris 2.5.1	**Linux 2.0.34**	**Solaris 2.5.1**	**Linux 2.0.34**	**FreeBSD 3.0**
ext2/ufs/ffs	1242.3	1097.0	1070.3	524.2	551.2
lofs	1251.2	1110.1	1081.8	530.6	n/a
wrapfs	1310.6	1148.4	1138.8	559.8	667.6
cryptfs	1608.0	1258.0	1362.2	628.1	729.2
crypt-wrap	22.7%	9.5%	19.6%	12.2%	9.2%
nfs	1490.8	1440.1	1374.4	772.3	689.0
cfs	2168.6	1486.1	1946.8	839.8	827.3
cfs-nfs	45.5%	3.2%	41.6%	8.7%	20.1%
crypt-cfs	34.9%	18.1%	42.9%	33.7%	13.5%

Table 3: Time (in seconds) to build a large package on various file systems and platforms. The percentage lines show the overhead difference between some file systems

First we evaluate the performance impact of stacking a file system. Lofs is 0.7–1.2% slower than the native disk based file system. Wrapfs adds an overhead of 4.7–6.8% for Solaris and Linux systems, but that is comparable to the 3–10% degradation previously reported for null-layer stackable file systems[6, 18]. On FreeBSD, however, Wrapfs adds an overhead of 21.1% compared to FFS: to overcome limitations in nullfs, we used synchronous writes. Wrapfs is more costly than lofs because it stacks over every vnode and keeps its own copies of data, while lofs stacks only on directory vnodes, and passes all other vnode operations to the lower level verbatim.

5.2 Cryptfs

Using the same tests we did for Wrapfs, we measured the performance of Cryptfs and CFS[2]. CFS is a user level NFS-based encryption file system. The results are also summarized in Table 3, for which the standard deviation did not exceed 0.8% of the mean.

Wrapfs is the baseline for evaluating the performance impact of the encryption algorithm. The only difference between Wrapfs and Cryptfs is that the latter encrypts and decrypts data and file names. The line marked as "*crypt-wrap*" in Table 3 shows that percentage difference between Cryptfs and Wrapfs for each platform. Cryptfs adds an overhead of 9.2–22.7% over Wrapfs. That significant overhead is unavoidable. It is the cost of the Blowfish cipher, which, while designed to be fast, is still CPU intensive.

[5] All machines used in these tests had 32MB RAM.

Measuring the encryption overhead of CFS was more difficult. CFS is implemented as a user-level NFS file server, and we also ran it using Blowfish. We expected CFS to run slower due to the number of additional context switches that it incurs and due to NFS v.2 protocol overheads such as synchronous writes. CFS does *not* use the NFS server code of the given operating system; it serves user requests directly to the kernel. Since NFS server code is implemented in general inside the kernel, it means that the difference between CFS and NFS is not just due to encryption, but also due to context switches. The NFS server in Linux 2.0 is implemented at user-level, and is thus also affected by context switching overheads. If we ignore the implementation differences between CFS and Linux's NFS, and just compare their performance, we see that CFS is 3.2–8.7% slower than NFS on Linux. This is likely to be the overhead of the encryption in CFS. That overhead is somewhat smaller than the encryption overhead of Cryptfs because CFS is more optimized than our Cryptfs prototype: CFS precomputes large stream ciphers for its encrypted directories.

We performed microbenchmarks on the file systems listed in Table 3 (reading and writing small and large files). These tests isolate the performance differences for specific file system operations. They show that Cryptfs is anywhere from 43% to an order of magnitude faster than CFS. Since the encryption overhead is roughly 3.2–22.7%, we can assume that rest of the difference comes from the reduction in number of context switches. Details of these additional measurements are available elsewhere[24].

5.3 Usenetfs

We configured a News server consisting of a Pentium-II 333Mhz, with 64MB of RAM, and a 4GB fast SCSI disk for the news spool. The machine ran Linux 2.0.34 with our Usenetfs. We created directories with exponentially increasing numbers of files in each: 1, 2, 4, etc. The largest directory had 524288 (2^{19}) files numbered starting with 1. Each file was 2048 bytes long. This size is the most common article size on our production news server. We created two hierarchies with increasing numbers of articles in different directories: one flat and one managed by Usenetfs.

We designed our next tests to match the two actions most commonly undertaken by a news server (see Table 2). First, a news server looks up and reads articles, mostly in response to users reading news and when processing outgoing feeds. The more users there are, the more random the article numbers read tend to be. While users read articles in a mostly sequential order, the use of threaded newsreaders results in more random reading. The (log-log) plot of Figure 5 shows the performance of 1000 random lookups in both flat and Usenetfs-managed directories. The times reported are in milliseconds spent by the process and the operating system on its behalf. For random lookups on directories with fewer than 1000–2000 articles, Usenetfs

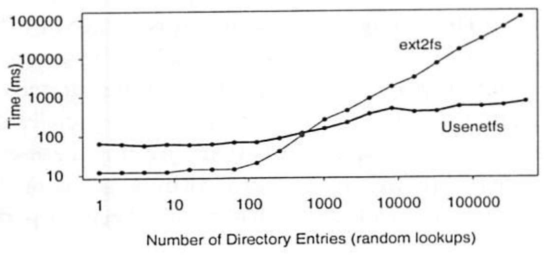


Figure 5: Cost for 1000 Random Article Lookups

adds overhead and slows performance. We expected this because the bushier directory structure Usenetfs maintains has over 1000 subdirectories. As directory sizes increase, lookups on flat directories become linearly more expensive while taking an almost constant time on Usenetfs-managed directories. The difference exceeds an order of magnitude for directories with over 10,000 articles.

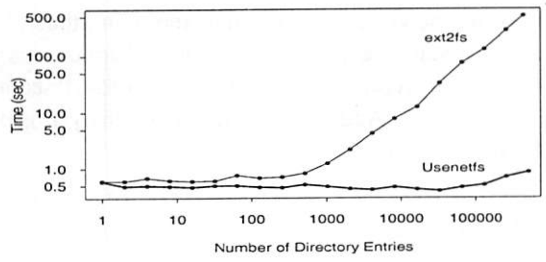


Figure 6: Cost for 1000 Article Additions and Deletions

The second common action a news server performs is creating new articles and deleting expired ones. New articles are created with monotonically increasing numbers. Expired articles are likely to have the smallest numbers so we made that assumption for the purpose of testing. Figure 6 (also log-log) shows the time it took to add 1000 new articles and then remove the 1000 oldest articles for successively increasing directory sizes. The results are more striking here: Usenetfs times are almost constant throughout, while adding and deleting files in flat directories took linearly increasing times.

Creating over 1000 additional directories adds overhead to file system operations that need to read whole directories, especially the readdir call. The last Usenetfs test takes into account all of the above factors, and was performed on our departmental production news server. A simple yet realistic measure of the overall performance of the system is to test how much reserve capacity was left in the server. We tested that by running a repeated set of compilations of a large package (Am-utils), timing how long it took to complete each build. We measured the compile times of Am-utils, once when the news server was running without Usenetfs management, and then when Usenetfs managed the top 6 newsgroups. The results are depicted in Figure 7. The average compile time was reduced by 22% from 243 seconds to 200 seconds. The largest savings appeared during busy times when our server transferred outgoing articles to our upstream feeds, and especially during the four daily expiration periods. During these expiration peaks, performance improved by a factor of 2–3. The overall effect of Usenetfs had been to keep the performance of the news server more flat, removing those load surges. The standard deviation for the compiles was reduced from 82 seconds (34% of the mean) to 61 seconds (29% of the mean). Additional performance analysis is provided elsewhere[23].

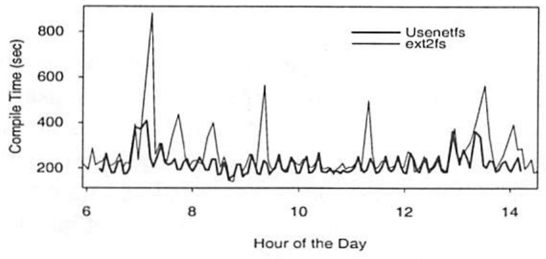


Figure 7: Compile Times on a Production News Server

5.4 Lb2fs

Lb2fs's performance is less than 5% slower than Wrapfs. The two main differences between Wrapfs and Lb2fs are the random selection algorithm and looking up directory vnodes on both replicas. The impact of the random selection algorithm is negligible, as it picks the least-significant bit of an internal system clock. The impact of looking up directory vnodes twice is bound by the ratio of directories to non-directories in common shared file systems. We performed tests at our department and found that the number of directories in such file systems to be 2–5% of the overall number of files. That explains the small degradation in performance of Lb2fs compared to Wrapfs.

5.5 Portability

We first developed Wrapfs and Cryptfs on Solaris 2.5.1. As seen in Table 4, it took us almost a year to fully develop Wrapfs and Cryptfs together for Solaris, during which time we had to overcome our lack of experience with Solaris kernel internals and the principles of stackable file systems. As we gained experience, the time to port the same file system to a new operating system grew significantly shorter. Developing these file systems for Linux 2.0 was a matter of days to a couple of weeks. This port would have been faster had it not been for Linux's different vnode interface.

File System	Solaris 2.x	Linux 2.0	FreeBSD 3.0	Linux 2.1
wrapfs	9 months	2 weeks	5 days	1 week
snoopfs	1 hour	1 hour	1 hour	1 hour
lb2fs	2 hours	2 hours	2 hours	2 hours
usenetfs		4 days		1 day
cryptfs	3 months	1 week	2 days	1 day

Table 4: Time to Develop and Port File Systems

The FreeBSD 3.0 port was even faster. This was due to many similarities between the vnode interfaces of Solaris and FreeBSD. We recently also completed these ports to the Linux 2.1 kernel. The Linux 2.1 vnode interface made significant changes to the 2.0 kernel, which is why we list it as another porting effort. We held off on this port until the kernel became more stable (only recently).

Another metric of the effort involved in porting Wrapfs is the size of the code. Table 5 shows the total number of source lines for Wrapfs, and breaks it down to three categories: common code that needs no porting, code that is easy to port by simple inspection of system headers, and code that is difficult to port. The hard-to-port code accounts for more than two-thirds of the total and is the one involving the implementation of each Vnode/VFS operation (operating system specific).

Porting Difficulty	Solaris 2.x	Linux 2.0	FreeBSD 3.0	Linux 2.1
Hard	80%	88%	69%	79%
Easy	15%	7%	26%	10%
None	5%	3%	5%	11%
Total Lines	3431	2157	2882	3279

Table 5: Wrapfs Code Size and Porting Difficulty

The difficulty of porting file systems written using Wrapfs depends on several factors. If plain C code is used in the Wrapfs API routines, the porting effort is minimal or none. Wrapfs, however, does not restrict the user from calling any in-kernel operating system specific function. Calling such functions complicates portability.

6 Related Work

Vnode stacking was first implemented by Rosenthal (in SunOS 4.1) around 1990[15]. A few other works followed Rosenthal, such as further prototypes for extensible file systems in SunOS[18], and the Ficus layered file system[4, 7] at UCLA. Webber implemented file system interface extensions that allow user level file servers[20]. Unfortunately this work required modifications to existing file systems and could not perform as well as in-kernel file systems.

Several newer operating systems offer a stackable file system interface. They have the potential of easy development of file systems offering a wide range of services.

Their main disadvantages are that they are not portable enough, not sufficiently developed or stable, or they are not available for common use. Also, new operating systems with new file system interfaces are not likely to perform as well as ones that are several years older.

The *Herd of Unix-Replacing Daemons* (HURD) from the Free Software Foundation (FSF) is a set of servers running on the Mach 3.0 microkernel[1] that collectively provide a Unix-like environment. HURD file systems run at user level. HURD introduced the concept of a translator, a program that can be attached to a pathname and perform specialized services when that pathname is accessed. Writing a new translator is a matter of implementing a well defined file access interface and filling in such operations as opening files, looking up file names, creating directories, etc.

Spring is an object-oriented research operating system built by Sun Microsystems Laboratories[10]. It was designed as a set of cooperating servers on top of a microkernel. Spring provides several generic modules which offer services useful for a file system: caching, coherency, I/O, memory mapping, object naming, and security. Writing a file system for Spring entails defining the operations to be applied on the objects. Operations not defined are inherited from their parent object. One work that resulted from Spring is the Solaris MC (Multi-Computer) File System[8]. It borrowed the object-oriented interfaces from Spring and integrated them with the existing Solaris vnode interface to provide a distributed file system infrastructure through a special *Proxy File System*. Solaris MC provides all of the benefits that come with Spring, while requiring little or no change to existing file systems; those can be gradually ported over time. Solaris MC was designed to perform well in a closely coupled cluster environment (not a general network) and requires high performance networks and nodes.

7 Conclusions

Wrapfs and the examples here prove that useful, non-trivial vnode stackable file systems can be implemented on modern operating systems without having to change the rest of the system. We achieve better performance by running the file systems in the kernel instead of at user-level. File systems built from Wrapfs are more portable than other kernel-based file systems because they interact directly with a (mostly) standard vnode interface.

Most complications discovered while developing Wrapfs stemmed from two problems. First, the vnode interface is not self-contained; the VM system, for example, offers memory mapped files, but to properly handle them we had to manipulate lower level file systems and MMU/TLB hardware. Second, several vnode calls (such as `readdir`) are poorly designed.

Estimating the complexity of software is a difficult task. Kernel development in particular is slow and costly because of the hostile development environment. Furthermore, personal experience of the developers figure heavily in the cost of development and testing of file systems. Nevertheless, it is our assertion that once Wrapfs is ported to a new operating system, other non-trivial file systems built from it can be prototyped in a matter of hours or days. We estimate that Wrapfs can be ported to any operating system in less than one month, as long as it has a vnode interface that provides a private opaque field for each of the major data structures of the file system. In comparison, traditional file system development often takes a few months to several years.

Wrapfs saves developers from dealing with kernel internals, and allows them to concentrate on the specifics of the file system they are developing. We hope that with Wrapfs, other developers could prototype new file systems to try new ideas, develop fully working ones, and port them to various operating systems—bringing the complexity of file system development down to the level of common user-level software.

We believe that a truly stackable file system interface could significantly improve portability, especially if adopted by the main Unix vendors. We think that Spring[10] has a very suitable interface. If that interface becomes popular, it might result in the development of many practical file systems.

7.1 Future

We would like to add to Wrapfs an API for manipulating file attributes. We did not deem it important for the initial implementation because we were able to manipulate the attributes needed in one place anyway.

Wrapfs cannot properly handle file systems that change the size of the file data, such as with compression, because these change file offsets. Such a file system may have to arbitrarily shift data bytes making it difficult to manipulate the file in fixed data chunks. We considered several designs, but did not implement any, because they would have complicated Wrapfs's code too much, and would mostly benefit compression.

8 Acknowledgments

The authors thank the anonymous reviewers and especially Keith Smith, whose comments improved this paper significantly. We would also like to thank Fred Korz, Seth Robertson, Jerry Altzman, and especially Dan Duchamp for their help in reviewing this paper and offering concrete suggestions. This work was partially made possible by NSF infrastructure grants numbers CDA-90-24735 and CDA-96-25374.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. *USENIX Conf. Proc.*, pages 93–112, Summer 1986.

[2] M. Blaze. A Cryptographic File System for Unix. *Proc. of the first ACM Conf. on Computer and Communications Security*, November 1993.

[3] B. Callaghan and S. Singh. The Autofs Automounter. *USENIX Conf. Proc.*, pages 59–68, Summer 1993.

[4] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. *USENIX Conf. Proc.*, pages 63–71, June 1990.

[5] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. *Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[6] J. S. Heidemann and G. J. Popek. File System Development with Stackable Layers. *ACM Transactions on Computing Systems*, **12**(1):58–89, February 1994.

[7] J. S. Heidemann and G. J. Popek. A layered approach to file system development. Technical report CSD-910007. University of California, Los Angeles, March 1991.

[8] V. Matena, Y. A. Khalidi, and K. Shirriff. Solaris MC File System Framework. Technical Report TR-96-57. Sun Labs, October 1996.

[9] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–97, August 1984.

[10] J. G. Mitchel, J. J. Giobbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An Overview of the Spring System. *CompCon Conf. Proc.*, 1994.

[11] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. *USENIX Conf. Proc.*, pages 137–52, June 1994.

[12] J. S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-Lite. *USENIX Conf. Proc.*, pages 25–33, January 1995.

[13] J. S. Pendry and N. Williams. Amd – The 4.4 BSD Automounter. User Manual, edition 5.3 alpha. March 1991.

[14] D. S. H. Rosenthal. Requirements for a "Stacking" Vnode/VFS Interface. Unix International document SD-01-02-N014. 1992.

[15] D. S. H. Rosenthal. Evolving the Vnode Interface. *USENIX Conf. Proc.*, pages 107–18, Summer 1990.

[16] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. *USENIX Conf. Proc.*, pages 119–30, June 1985.

[17] B. Schneier. Blowfish. In *Applied Cryptography, Second Edition*, pages 336–9. John Wiley & Sons, 1996.

[18] G. C. Skinner and T. K. Wong. "Stacking" Vnodes: A Progress Report. *USENIX Conf. Proc.*, pages 161–74, Summer 1993.

[19] SMCC. lofs – loopback virtual file system. SunOS 5.5.1 Reference Manual, Section 7. March 1992.

[20] N. Webber. Operating System Support for Portable Filesystem Extensions. *USENIX Conf. Proc.*, pages 219–25, Winter 1993.

[21] E. Zadok. *FiST: A File System Component Compiler*. PhD thesis, published as Technical Report CUCS-033-97. Computer Science Department, Columbia University, April 1997.

[22] E. Zadok. Am-utils (4.4BSD Automounter Utilities). Am-utils version 6.0a16 User Manual. April 1998. Available http://www.cs.columbia.edu/~ezk/am-utils/.

[23] E. Zadok and I. Badulescu. Usenetfs: A Stackable File System for Large Article Directories. Technical Report CUCS-022-98. Computer Science Department, Columbia University, June 1998.

[24] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A Stackable Vnode Level Encryption File System. Technical Report CUCS-021-98. Computer Science Department, Columbia University, July 1998.

9 Author Information

Erez Zadok is an Ph.D. candidate in the Computer Science Department at Columbia University. He received his B.S. in Comp. Sci. in 1991, and his M.S. degree in 1994, both from Columbia University. His primary interests include file systems, operating systems, networking, and security. The work described in this paper was first mentioned in his Ph.D. thesis proposal[21].

Ion Badulescu holds a B.A. from Columbia University. His primary interests include operating systems, networking, compilers, and languages.

Alex Shender is the manager of the computer facilities at Columbia University's Computer Science Department. His primary interests include operating systems, networks, and system administration. In May 1998 he received his B.S. in Comp. Sci. from Columbia's School of Engineering and Applied Science.

For access to sources for the file systems described in this paper see *http://www.cs.columbia.edu/~ezk/research/software/*.

Why does file system prefetching work?

Elizabeth Shriver
Information Sciences Research Center
Bell Labs, Lucent Technologies
shriver@research.bell-labs.com

Christopher Small
Information Sciences Research Center
Bell Labs, Lucent Technologies
chris@research.bell-labs.com

Keith A. Smith
Harvard University
keith@eecs.harvard.edu

Abstract

Most file systems attempt to predict which disk blocks will be needed in the near future and prefetch them into memory; this technique can improve application throughput as much as 50%. But why? The reasons include that the disk cache comes into play, the device driver amortizes the fixed cost of an I/O operation over a larger amount of data, total disk seek time can be decreased, and that programs can overlap computation and I/O. However, intuition does not tell us the relative benefit of each of these causes, or techniques for increasing the effectiveness of prefetching.

To answer these questions, we constructed an analytic performance model for file system reads. The model is based on a 4.4BSD-derived file system, and parameterized by the access patterns of the files, layout of files on disk, and the design characteristics of the file system and of the underlying disk. We then validated the model against several simple workloads; the predictions of our model were typically within 4% of measured values, and differed at most by 9% from measured values. Using the model and experiments, we explain why and when prefetching works, and make proposals for how to tune file system and disk parameters to improve overall system throughput.

1 Introduction

Previous work has shown that most file reads are sequential [Baker91]. Optimizing for the common case, modern file systems take advantage of this fact and prefetch blocks from disk that have not yet been requested, but are likely to be needed in the near future. This technique is effective for several reasons:

- There is a fixed cost associated with performing a disk I/O operation. By increasing the amount of data transfered on each I/O, the overhead is amortized over a larger amount of data, improving overall performance.
- Modern disks contain a disk cache, which contains some number of disk blocks from the cylinders of recent requests. If multiple blocks are read from the same track, all but the first may, under certain circumstances, be satisfied by the disk cache without accessing the disk surface.
- The device driver or disk controller can sort disk requests to minimize the total amount of disk head positioning done. With a larger list of disk requests, the driver or controller can do a better job of ordering them to minimize disk head motion. Additionally, the blocks of a file are often clustered together on the disk; when this is so, multiple blocks of the file can be read at once without an intervening seek.
- If the application performs substantial computation as well as I/O, prefetching may allow the application to overlap the two, which would increase the application's throughput. For example, an MPEG player may spend as much time computing as it does waiting for I/O; if the file system can run ahead of the MPEG player, loading data into memory before they are needed, the player will not block on I/O.

If an application spends as much time performing I/O as it does computing—successfully prefetching data will allow overlapping the two and the application's throughput will double.

This paper presents and validates an analytic file system performance model that allows us to explain why and when prefetching works, and makes recommendations for how to improve the benefit of file prefetching. The model described here is restricted in two ways.

First, it addresses only file read traffic, and does not capture file writes or file name lookup. Although this restricts the applicability of the model, we believe that there are many interesting workloads covered by the model. For example, many workloads consist almost entirely of reads, such as web servers, read-mostly database systems, and file systems that store programs, libraries, and configuration files (e.g., /, /etc, and /usr). Additionally, studies have shown that, for some engineering workloads, 80% of file requests are reads [Baker91, Hartman93].

It has long been the case that in order to improve performance, file systems use a *write-back* cache, delaying writes for some period of time [Feiertag72, Ritchie78, McKusick84]. The results of the study by Baker and associates showed that with a 30-second write-back delay, 36% to 63% of the bytes written to the cache do not survive, and by increasing the write-back delay size to 1000 seconds, 60% to 95% do not survive.

Second, although our model includes multiple concurrent programs accessing the file system, we limit it to workloads where each file is read from start to finish with little or no *think time* between read requests. Although this does not cover workloads such as the MPEG player discussed above, we believe that, given the relative speed of modern processors and I/O devices, in the common case think time is immeasurable. The Baker study showed that 75% of files were open less than a quarter of a second [Baker91]. This study was done on a collection of machines running at around 25 MHz (SPARCStation 1, Sun 3, DECStation 3100, DECStation 5000); on modern machines, with clock speeds an order of magnitude faster, think time should be substantially lower.

Outline of paper. Section 2 discusses in detail the modeled file system (the 4.4BSD Fast File System) and disk. Section 3 presents previous work in file systems and prefetching policies. The analytic model of file system response time and its validation are presented in Section 4 and Appendix A. Section 5 explains why and when prefetching works. We summarize our work and discuss future work in Section 6.

2 Background

In this section we describe in detail the behavior of the file system we model, and discuss the characteristics of modern disks.

2.1 The file system

The file system that we used as the basis for our model is the 4.4BSD implementation of the Berkeley Fast File System that ships with BSD/OS 3.1 [McKusick96].

Reading files. An application can make a request for an arbitrarily large amount of data from a file. To process an application-level read request, the file system divides the request into one or more block-sized (and block-aligned) requests, each of which the file system services separately. A popular file system block size is 8 KB, although other block sizes can be specified when the file system is initialized.

For each block in the request, the file system first determines whether the block is already in the operating system's in-memory cache. If it is, then the block is copied from the cache to the application. If the block is not already in memory, the file system issues a read request to the disk device driver.

Regardless of whether the desired block is already in memory, the file system may prefetch one or more subsequent blocks from the file. The amount of data the file system prefetches is determined by the file system's prefetch policy, and is a function of the current file offset and whether or not the application has been accessing the file sequentially. A read of block x from a file is *sequential* if the last block read from that file was either x or $x - 1$. By treating successive reads of the same block as "sequential," applications are not penalized for using a read size that is less than the file system's block size.

As read requests are synchronous, the operating system blocks the application until all of the data it has requested are available. Note that a single disk request may span multiple blocks and include both the requested data and prefetch data, in which case the application can not continue until the entire request completes.

Data placement on disk. A *cluster* is a group of logically sequential file blocks that are stored sequentially on disk; the *cluster size* is the number of bytes in the cluster. Depending on the file system parameters, the file system may place successive allocations of clusters contiguously on the disk. This can result in contiguous allocations of hundreds of kilobytes in size.

The blocks of a file are indexed by a tree structure on disk; the root of the tree is an *inode.* The inode contains the disk addresses to the first few blocks of a file (i.e., the first DirectBlocks blocks of the file); in the case of the modeled system, the inode contains pointers to the first twelve blocks. The remaining blocks are referenced by *indirect blocks.*

The first block referenced from an indirect block is always the start of a new cluster. This may cause the preceding cluster to be smaller than the file system's cluster size. For example, if DirectBlocks is not a multiple of the cluster size, the last cluster of direct blocks may be smaller than the cluster size.

The file system divides the disk into cylinder groups, which are used as allocation pools. Each cylinder group contains a fixed sized number of blocks (2048 blocks, or 16 MB on the modeled system). The file system exploits expected patterns of locality of reference by co-locating related data in the same cylinder group.

The file system usually attempts to allocate clusters for the same file in the same cylinder group. Each cluster is allocated in the same cylinder group as the previous cluster. The file system attempts to space clusters according to the value of the rotational delay parameter which is set using the `newfs` or `tunefs` command. The file system can always achieve this spacing on an empty file system. If the free space on the file system is fragmented, however, this spacing may vary. The file system allocates the first cluster of a file from the same cylinder group as the file's inode. Whenever an indirect block is allocated to a file, allocation for the file switches to a different cylinder group. Thus an indirect block and the clusters it references are allocated in a different cylinder group than the previous part of the file.

Prefetching in the file system cache. If the requested data are not in cache, the file system issues a disk request for the desired block. If the application is accessing the file sequentially, the file system may prefetch one or more additional data blocks. The amount of data prefetched is doubled on each disk read, up to a maximum of the cluster size. The last block of a file may be allocated to a *fragment* rather than a full size block. When this happens, the final fragment of the file is not prefetched.

It is possible for a block to be prefetched and then evicted before it is requested. If the user subsequently requests such a block, the file system assumes that it is prefetching too aggressively and cuts the prefetch size in half.

2.2 The disk

When a disk request is issued from the file system, it enters the device driver. If the disk is busy, the request is put on a queue in the device driver; the queue is sorted by a scheduling algorithm that attempts to improve response times. One commonly-used class of scheduling algorithms are the *elevator* algorithms, where the requests are serviced in the order that they appear on the disk tracks. CLOOK and CSCAN are examples of elevator algorithms. Once the request reaches the head of the queue, the request is sent to the bus controller which gains control of the bus. The request is then sent to the disk, and might be queued there if the disk mechanism is busy. This queue is also sorted to improve response time; one commonly-used scheduling algorithm is Shortest Positioning Time First, which services requests in an order intended to minimize the sum of the *seek time* (i.e., the time to move the head from the current track to the desired track) and the *rotational latency* (i.e., the time needed for the disk to rotate to the correct sector once the desired track is reached).

Seek time. Seek is the time for the actuator to move the disk arm to the desired cylinder. A seek operation can be decomposed into:

- *speedup*, where the arm is accelerated until it reaches half of the seek distance or a fixed maximum velocity,
- *coast* for long seeks, where the arm is moving at maximum velocity,
- *slowdown*, where the arm is brought to rest close to the desired track, and
- *settle*, where the disk controller adjusts the head to access the desired location.

Very short seeks (2–4 cylinders) are dominated by the settle time. Short seeks (less than 200–400 cylinders) are dominated by the speedup, which is proportional to the square root of seek distance. Long seeks are dominated by the coast time, which is proportional to the seek distance. Thus, the seek time can be approximated by a function such as

$$\mathsf{Seek\_Time[dis]} = \begin{cases} 0 & \mathsf{dis} = 0 \\ a + b\sqrt{\mathsf{dis}} & 0 < \mathsf{dis} \leq e \\ c + d\,\mathsf{dis} & \mathsf{dis} > e \end{cases} \qquad (1)$$

where a, b, c, d, and e are device-specific parameters and dis is the number of cylinders to be traveled. Single cylinder seeks are often treated specially.

The disk cache. When the request reaches the head of the queue, the disk checks its cache to see if the data are in cache. If not, the disk mechanism moves the disk head to the desired track (seeking) and waits until the desired sector is under the head (rotational latency). The disk then reads the desired data into the disk cache. The disk controller then contends for access to the bus, and transfers the data to the host from the disk cache at a rate determined by the speed of the bus controller and the bus itself. Once the host receives the data and copies them into the memory space of the file system, the system awakens any processes that are waiting for the read to complete.

The disk cache is used for multiple purposes. One is as a pass-through speed-matching buffer between the disk mechanism and the bus. Most disks do not retain data in the cache after the data have been sent to the host. A second purpose is as a readahead buffer. Data can be readahead into the disk cache to service future requests. Most frequently, this is done by the disk saving in a cache segment the data that comes after the requested data. Modern disks such as the Seagate Cheetah only readahead data when the requested addresses suggest that a sequential access pattern is present.

The disk cache is divided into *cache segments*. Each segment contains data prefetched from the disk for one sequential stream. The number of cache segments usually can be set on a per-disk basis; the typical range of allowable values is between one and sixteen.

Disk performance is covered in more detail by Shriver [Shriver97] and by Ruemmler and Wilkes [Ruemmler94].

3 Related work

Prefetching. Prefetching is not a new idea; in the 1970's, Multics supported prefetching [Feiertag72], as did Unix [Ritchie78]. Earlier work has focused on the benefit of prefetching, either by allowing applications to give prefetching hints to the operating system [Cao94, Patterson95, Mowry96], or by automatically discovering file access patterns in order to better predict which blocks to prefetch [Griffioen94, Lei97, Kroeger96]. Techniques studied have included neural networks [Madhyastha97a] and hidden Markov models [Madhyastha97b]. Our work differs from this work in three ways. First, we address only common case workloads that have sequential access patterns. Second, our model is parameterized by the file system's behavior such as caching strategy and file layout, and takes into account the behavioral characteristics of the disks used to store files. Third, our model predicts the performance of the file system.

Substantial work has been done studying the interaction between prefetching and caching [Cao95, Patterson95, Kimbrel96]. Others have examined methods to work around the file system cache to achieve the desired performance (e.g., [Kotz95]).

The benefit of prefetching is not limited to workloads where files are read sequentially; Small studied the effect of prefetching on random-access, zero think time workloads on the VINO operating system, and showed that even with these workloads the performance gain from prefetching was more than 20% [Small98].

Disk modeling. Much work has been done in disk modeling. The usual approach to analyzing detailed disk drive performance is to use simulation (e.g., [Hofri80, Seltzer90, Worthington94]). Most early modeling studies (e.g., [Bastian82, Wilhelm77]) concentrated on rotational position sensing for mainframe disk drives, which had no cache at the disk and did no readahead. Most prior work has not been workload specific, and has, for example, assumed that the workload has uniform random spatial distributions (e.g., [Seeger96, Ng91, Merchant96]). Chen and Towsley, and Kuratti and Sanders, modeled the probability that no seek was needed [Chen93, Kuratti95]; Hospodor reported that an exponential distribution of seek times matched measurements well for three test workloads [Hospodor95]. Shriver and colleagues, and Barve and associates present analytic models for modern disk drives, representing readahead and queueing effects across a range of workloads [Shriver97, Shriver98, Barve99].

4 The analytic model

In this section, we present the file system, disk, and workload parameters that we need for our model. As we present the needed file system and disk parameters, we also give the values for the platforms which we used to validate the model. We close this section with presenting the details of the model.

We used two platforms; one with a slow bus (i.e., 10 MB/s), and one with a fast bus (i.e., 20 MB/s). Details of our test/experiment platforms are in Section 5.

4.1 File system specification

Based on our understanding of the file system cache policies, we determined a set of parameters that allow us to capture the performance of the file system cache; these can be found in Table 1. For our fast machine, the SystemCallOverhead value was 5 μs and the MemoryCopyRate was 5 μs/KB.

4.2 Disk specification

To predict the disk response time, we need to know several parameters of the disk being used.

- DiskOverhead includes the time to send the request down the bus and the processing time at the controller, which is made up of the time required for the controller to parse the request, check the disk cache for the data, and so on. DiskOverhead can either be approximated using a complex disk model [Shriver97] or can be measured experimentally. In this paper we measured the disk overhead experimentally at 1.8 ms for a single file and 1.2 ms for multiple files for our slow platform and 0.34 ms for our fast platform.
- *seek curve information* is used to approximate the seek time. The seek curve information we use is $a = 0.002$, $b = 0.173$, $c = 3.597$, $d = 0.002$, and $e = 801$ as defined in equation (1).
- *disk rotation speed* is used to approximate the time spent in rotational latency. The DiskTR is the rate that data can be transfered from the disk surface to the disk cache. The disk used to validate our model spins at 10,000 RPM, giving us a DiskTR of close to 18 MB/s.
- BusTR gives us the rate at which data can be transfered from the disk cache to the host; we are bounded by the slower of the BusTR and DiskTR. On the slow platform, the transfer rate was limited to 9.3 MB/s; on the fast platform, the transfer rate was 18.2 MB/s.
- CacheSegments is the number of different data streams the disk can concurrently cache, and hence the number of streams for which it can perform read-ahead. The disk used to validate our model was configured for three cache segments; this model of disk can be configured for between one and sixteen cache segments.
- CacheSize is the size of the disk cache. From this value and the CacheSegments, the size of each cache segment can be computed. The disk used to validate our model has a 512 KB cache.
- Max_Cylinder is the number of cylinders in the disk. The disk used to validate our model has 6526 cylinders.

Table 1: File system parameters and values for validated platform.

parameter	*definition*	*validated platform*
BlockSize	the amount of data which the file system processes at once	8 KB
DirectBlocks	the number of blocks that can be accessed before the indirect block needs to be accessed	12
ClusterSize	the amount of a file that is stored contiguously on disk	64 KB
CylinderGroupSize	number of bytes on a disk that file system treats as "close"	16 MB
SystemCallOverhead	time needed to check the file system cache for the requested data	10 μs
MemoryCopyRate	rate at which data are copied from the file system cache to the application memory	10 μs/KB

4.3 Workload specification

The workload parameters that affect file system cache performance are the ones needed to predict the disk performance and the file layout on disk. Table 2 presents this set of parameters; most of these parameters were taken from earlier work on disk modeling [Shriver98].[1]

4.4 The model

Our approach has been to use the technique presented in our earlier work on disk modeling, which models the individual components of the I/O path, and then composes the models together [Shriver97]. We use some of the ideas presented in the disk cache model to model the file system cache.

Disk response time. The mean disk response time is the sum of disk overhead, disk head positioning time, and time to transfer the data from disk to the file system cache:

$$\mathsf{DRT} = \mathsf{DiskOverhead} + \mathsf{PositionTime} + \mathbf{E}[\mathsf{disk\_request\_size}] / \min\{\mathsf{BusTR}, \mathsf{DiskTR}\}.$$

(Note: $\mathbf{E}[x]$ denotes the expected, or average value for x.) The amount of time spent positioning the disk head, PositionTime, depends on the current location of the disk head, which is determined by the previous request. For example, if this is the first request for a block in a given cluster, PositionTime will include both seek time and time for the rotational latency. Let $\mathbf{E}[\mathsf{SeekTime}]$ be the mean seek time and $\mathbf{E}[\mathsf{RotLat}]$ be the mean rotational latency (1/2 the time for a full disk rotation). Thus, the disk response time for the first request in a cluster is

$$\mathsf{DRT}[\text{random request}] = \mathsf{DiskOverhead} + \mathbf{E}[\mathsf{SeekTime}] + \mathbf{E}[\mathsf{RotLat}] + \frac{\mathbf{E}[\mathsf{disk\_request\_size}]}{\min\{\mathsf{BusTR}, \mathsf{DiskTR}\}}. \qquad (2)$$

If the previous request was for a block in the same cylinder group, the seek distance will be small. This will be the case if the previous read was to a portion of the file stored in the same cylinder group, or to some other file found in the same cylinder group. If there are n files being accessed concurrently, the expected seek distance will either be (a) Max_Cylinder/3, if the device driver and disk controller request queues are empty, or (b) (assuming the disk scheduler is using an elevator algorithm) Max_Cylinder/$(n+2)$ [Shriver97].

The mean disk request size, $\mathbf{E}[\mathsf{disk\_request\_size}]$, can be computed by averaging the request sizes; these can be computed by simulating the algorithm to determine the amount of data prefetched, where the simulation stops when the amount of accessed data is equal to ClusterSize. If the file system is servicing more than one file, the actual amount prefetched can be smaller than expected due to blocks being evicted before use. If the file system is not prefetching data, the $\mathbf{E}[\mathsf{disk\_request\_size}]$ is the file system block size, BlockSize.

Sometimes the requested data are in the disk cache due to readahead; in these cases, the disk response time is

$$\mathsf{DRT}[\text{cached request}] = \mathsf{DiskOverhead} + \mathbf{E}[\mathsf{disk\_request\_size}]/\mathsf{BusTR}. \qquad (3)$$

[1]The previous disk model includes additional workload parameters that support specification of spatial locality; these are not needed for our current model since we assume that the files are accessed sequentially. The earlier disk model also supports a read fraction parameter; in this paper, we only model file reads.

Table 2: Workload specification.

parameter	*definition*	*unit*
temporal locality measures		
request_rate	rate at which requests arrive at the storage device	requests/second
cylinder_group_id	cylinder group (location) of the file	integer
arrival_process	inter-request timing (constant [open, closed], Poisson, or bursty)	—
spatial locality measures		
data_span	the span (range) of data accessed	bytes
request_size	length of a host read or write request	bytes
run_length	length of a *run*, a contiguous set of requests	bytes

File system response time. We first compute the amount of time needed for all of the file system accesses TotalFSRT, and then compute the mean response time for each access, FSRT, by averaging:

$$\mathsf{FSRT} = \frac{\mathsf{data\_span}}{\mathsf{request\_size}} \mathsf{TotalFSRT}. \qquad (4)$$

The rest of this section discusses approximating TotalFSRT.

Let us first look at the simplest case: reading one file that resides entirely in one cluster, the mean response time to read the cluster contains file system overhead plus the time needed to access the data from disk:

$$\begin{aligned}\mathsf{ClusterRT} = {} & \mathsf{FSOverhead} + \\ & \mathsf{DRT}[\text{first request}] + \\ & \sum_i \mathsf{DRT}[\text{remaining request}_i]\end{aligned}$$

where the first request and remaining requests are the disk requests for the blocks in the cluster and DRT[first request] is from equation (2). If n files are being serviced at once, the DRT[remaining request$_i$]'s each contain $\mathbf{E}$[SeekTime] + $\mathbf{E}$[RotLat] if n is more than CacheSegments, the number of disk cache segments. If not, some of the data will be in disk cache and equation (3) is used. The FSOverhead can be measured experimently or computed as SystemCallOverhead + $\mathbf{E}$[request_size]/MemoryCopyRate. The number of requests per cluster can be computed as data_span/disk_request_size.

If the files span multiple clusters, we have

$$\mathsf{TotalFSRT} = \mathsf{NumClusters} \cdot \mathsf{ClusterRT}$$

where we approximate the number of clusters as NumClusters = data_span/ClusterSize. To capture the "extra" cluster due to only the first DirectBlocks blocks being stored on the same cluster, this value is incremented by 1 if (ClusterSize/BlockSize)/DirectBlocks is not 1 and data_span/BlockSize > DirectBlocks.

If the device driver or disk controller scheduling algorithm is CLOOK or CSCAN and the queue is not zero, then there is a large seek time (for CLOOK) or a full stroke seek time (for CSCAN) for each group of n accesses, when n is the number of files being serviced by the file system; we call this time extra_seek_time.

If the n files being read are larger than DirectBlocks, we must include the time required to read the indirect block:

$$\begin{aligned}\mathsf{TotalFSRT} = {} & n \cdot \mathsf{NumClusters} \cdot \mathsf{ClusterRT} + \\ & \mathsf{num\_requests} \cdot \mathsf{extra\_seek\_time} + \\ & \mathsf{DRT}[\text{indirect block}] \qquad (5)\end{aligned}$$

where num_requests is the number of disk requests in a file. Since the location of the indirect block is on a random cylinder group, equation (2) is used to compute DRT[indirect block]. Of course, if the file contains more blocks than can be referenced by both the inode and the indirect block, multiple indirect block terms are needed.

5 Discussion

In the introduction to this paper, we listed the reasons that prefetching improves performance: the disk cache comes into play, the device driver amortizes the fixed cost of an I/O over a larger amount of data, and total disk seek time can be decreased. In this section we discuss the terms introduced by our model and attempt to explain where the time goes, and when and why prefetching works. To do this, we collected detailed traces of a variety of workloads. These traces allowed us to compute the file system and disk response times experienced by the test machines. These response times were also used in our validations as discussed in Appendix A.

Hardware setup and trace gathering. We performed experiments on two hardware configurations: a 200 MHz Pentium Pro processor and a 450 MHz Pentium II processor. Each machine had 128 MB of main memory. We conducted all of our tracing and measurements on a 4 GB Seagate ST34501W (Cheetah) disk, connected via a 10 MB/second PCI SCSI controller (for the 200 MHz processor) or via a 20 MB/second PCI SCSI controller (for the 450 MHz processor). Our test machines were running version 3.1 of the BSD/OS operating system. The file system parameters for our test file systems are found in Table 1 and the disk parameters are in Section 4.2.

We collected our traces by adding trace points to the BSD/OS kernel. At each trace point the kernel wrote a record to an in-memory buffer describing the type of event that had occurred and any related data. The kernel added a time stamp to each trace record, using the CPU's on-chip cycle counter. A user-level process periodically read the contents of the trace buffer.

To document the response time for application-level read requests, we used a pair of trace points at the entry and exit of the FFS read routine. Similarly, we measured disk-level response times using a pair of trace points in the SCSI driver, one when a request is issued to the disk controller, and a second when the disk controller indicates that the I/O has completed. Additional trace points documented the exact sequence (and size) of the prefetch requests issued by the file system, and the amount of time each request spent on the operating system's disk queue.

The numbers discussed in this section are for the machine with the faster bus unless stated otherwise.

Disk cache. When an application makes a read request of the file system, the file system checks to see if the requested data are in its cache, and if not, issues an I/O request to the disk. The data will be found in the file system cache if they were prefetched

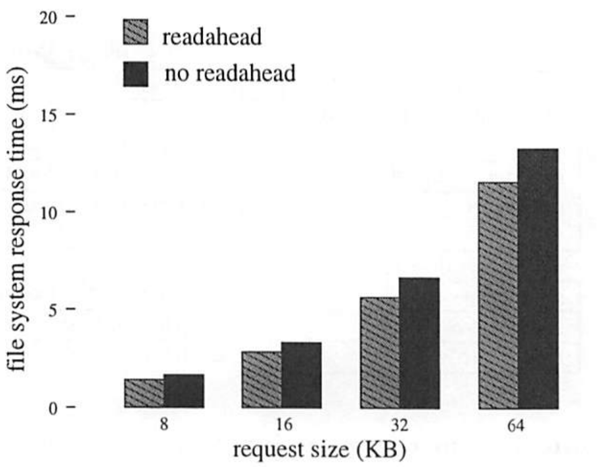


Figure 1: File system response times for a 64 KB file with and without the disk cache performing readahead. The readahead values are measured; the no-readahead values are predicted by the model.

and have not been evicted. If the data are not in the file system's cache, the file system must read it from the disk. There are two possible scenarios:

1. The data are in the disk cache as a result of readahead for a previous command, so the disk does not need to read the data again. The disk sends the data directly from the disk cache.

2. The data are not in the disk cache and must be read from the disk surface.

In an attempt to quantify the effect of the disk cache, Figures 1 and 2 contain the file system response time measurements with the disk cache performing readahead and file system response time predictions without the disk cache performing readahead. The percent improvement in the response time when the disk cache is performing readahead is 17–23%.

Modern disks are capable of caching data for concurrent workloads, where each workload is reading a region of the disk sequentially. If there are enough cache segments for the current number of sequential workloads, the disk will readahead for each workload, and each workload will benefit. However, if there are more sequential workloads than cache segments, depending on the cache replacement algorithm used by the disk, the disk's ability to prefetch may have little or no positive effect on performance. In addition, disk readahead is only valuable when the file system prefetch size is less than the cluster

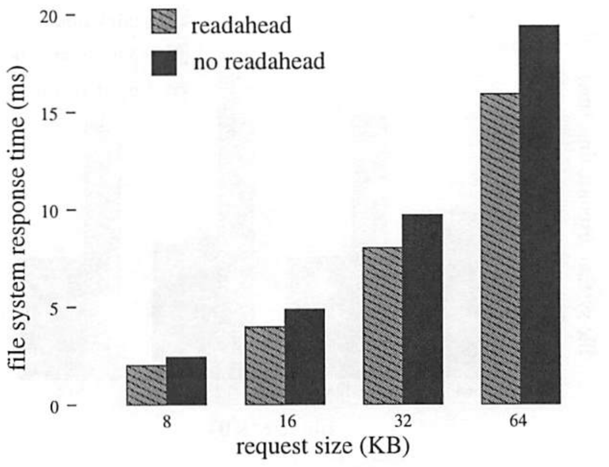


Figure 2: File system response times for a 128 KB file with and without the disk cache performing readahead. The readahead values are measured; the no-readahead values are predicted by the model.

size, since, after that, the entire cluster is fetched with one disk access. In the case of our file system parameters, this occurred when the file was 128 KB or smaller.

I/O cost amortization. On our slow bus configuration, we measured the disk overhead of performing an I/O operation at 1.2 to 1.8 ms, which is on the same order as the time to perform a half-rotation (3 ms). The measured transfer rate of the bus is 9.3 MB/s; by saving an I/O operation, we can transfer an additional 11 to 16 KB. As an example, assume that 64 KB of data will be used by the application. If the requested data are in the disk cache, using a file block of 8 KB will take at least 14.1 ms (1.8 ms overhead four times + 6.9 ms for data transfer);[2] a file block of 64 KB will take 8.7 ms (1.8 ms overhead + 6.9 ms for data transfer), just a little over half the I/O time.

The impact of I/O cost amortization can be seen when comparing the measured file system response time when servicing one file, with and without prefetching. Figure 3 show these times for the slower hardware configuration. With prefetching disabled, the file system requests data in BlockSize units, increasing the number of requests, and the amount of disk and file system overhead. The additional overheads increase the resulting performance by 13–29%.

[2]With the file system performing prefetching, there will be 4 disk requests having a mean disk request size of 16 KB.

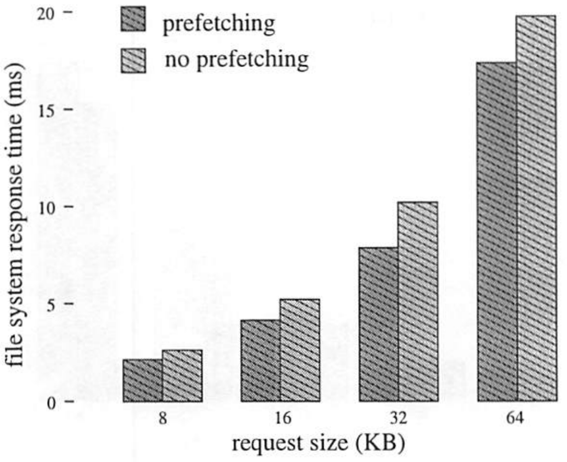


Figure 3: Measured file system response times for a 64 KB file with and without the file system performing prefetching.

When we ran our tests on the machine with the faster bus, we noted anomalous disk behavior that we do not yet understand. According to our measurements, it should (and does, in most cases) take roughly 0.78 ms to read an 8 KB block from the disk cache over the bus on this machine. However, under certain circumstances, 8 KB reads from the disk cache complete more quickly, taking roughly 0.56 ms.[3] This happened only when file system prefetching is disabled, i.e., when the file system requests multiple consecutive 8 KB blocks, and when there is only one application reading from the disk. The net result is that, in these rare situations, performance is slightly *better* with file system prefetching disabled. This behavior also was displayed with the slower bus, but as you can see in Figure 3, the bus is slow enough so that the response time with prefetching is smaller than the response time without prefetching.

Seek time reduction. As the number of active workloads increases, the latency for each workload will increase, but the disk throughput can, paradoxically, increase as well. Due to the type of scheduling algorithms used for the device driver queue, more elements in the read queue can mean smaller seeks between each read, and hence greater disk throughput. On the other hand, a longer queue means that each request will, on average, spend more time in the queue, and thus the read latency will be greater.

[3]We are in communication with Seagate in an attempt to determine why we are seeing this behavior.

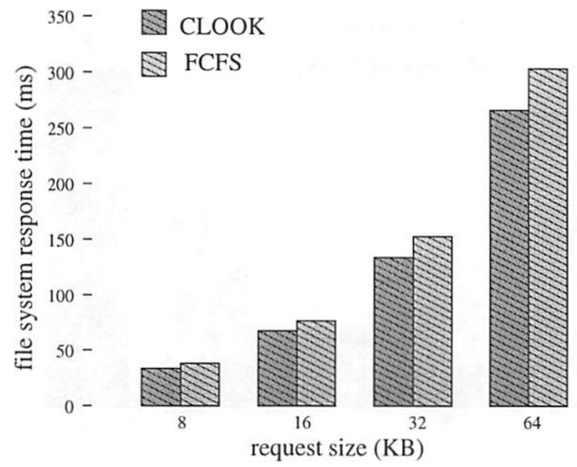


Figure 4: Measured file system response times for 8 64 KB files with the device driver using a CLOOK scheduling algorithm with a FCFS scheduling algorithm.

Figure 4 displays the file system response time with the device driver implementing the CLOOK scheduling algorithm (the standard algorithm), and implementing FCFS, which will not reduce the seek time. The performance gain from using CLOOK over FCFS is 14%.

I/O / computation overlap. As was discussed in Section 1, if an application performs substantial computation as well as I/O, prefetching may allow the application to overlap the two, increasing application throughput and decreasing file system response time. For example, on our test hardware, computing the MD5 checksum of a 10 KB block of data takes approximately one millisecond. A program reading data from the disk and computing the MD5 checksum will exhibit delays between successive read requests, giving the file system time to prefetch data in anticipation of the next read request. Figure 5 shows the file system response times with a request size of 10 KB for files of varying lengths. The figure shows the response time given no delay (representing the application having no I/O / computation overlap), with an application delay of 0.5 ms, and with an application delay of 1.0 ms (as with MD5). As the file size increases, so do the savings due to prefetching. With a 64 KB file there is a 36% improvement, compared to a 114% improvement when reading a 512 KB file.

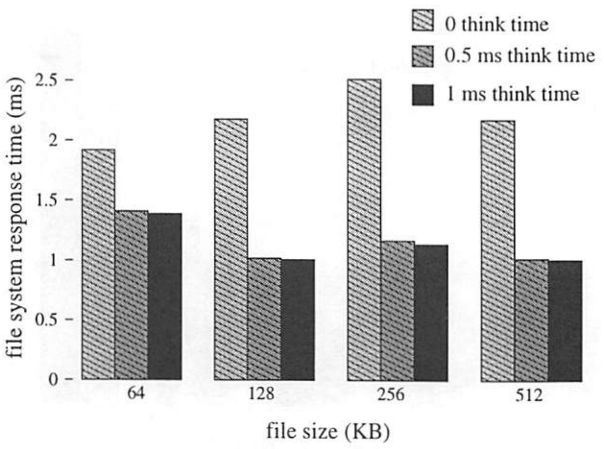


Figure 5: Measured file system response times with the application think time of 0 ms, 0.5 ms, and 1 ms.

Summary. In this section we showed how the components of prefetching combine to improve performance. In our tests, disk caching provided a 17–23% boost, I/O cost amortization yielded, 13–29%, seek time reduction (CLOOK *vs.* FCFS) 14%, and overlapping computation and I/O can save as much as 50%. Depending on the behavior of the specific application, these components can have a greater or lesser benefit; added together, the performance gain is significant.

6 Conclusions

We developed an analytic model that predicts the response time of the file system with a maximum relative error of 9% (see Appendix A). Given the wide range of conceivable file system layouts and prefetching policies, an accurate analytic model simplifies the task of setting system parameters that may improve performance enough to be worth implementing and studying in more detail.

Our model has allowed us to develop two suggestions for decreasing the file system response time. If it is reasonable to assume that the prefetched data will be used, and we have room in the file system cache, once the disk head has been positioned over a cluster, the entire cluster should be read. This will decrease the file system and disk overheads.

The number of disk cache segments restricts the number of sequential workloads for which the disk

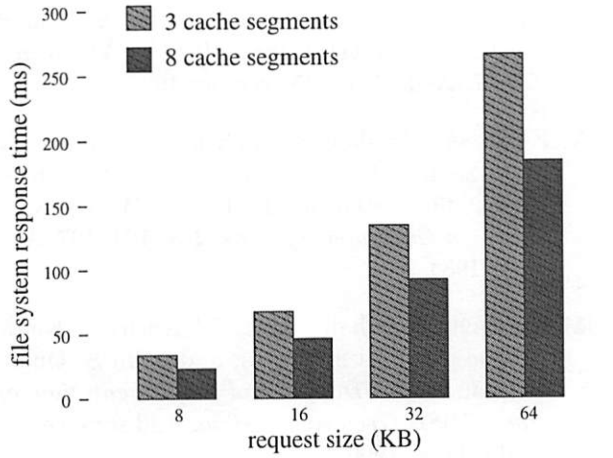


Figure 6: File system response times for 8 64 KB files with 3 and 8 segments. 3-segment values are measured and 8-segment values are predicted by the model.

cache can perform readahead, which means that if the number of disk cache segments is smaller than the number of concurrent workloads, it can be as if the disk had no cache at all. One enhancement that we suggest is for the file system to dynamically modify the number of disk cache segments to be the number of files being concurrently accessed from that disk. This is a simple and inexpensive SCSI operation, and can mean the difference between having the disk cache work effectively and having it not work at all. Figure 6 compares the measured file system response time when servicing 8 workloads with 3 cache segments and the predicted response time with 8 cache segments. This shows us a 44–46% decrease in the response time when the number of cache segments is set to the number of concurrent workloads.

Our current model does not handle file system writes; we would like to extend it to support writes. We would like to use our analytic model to compare different file system prefetching polices and parameter settings to determine the "best" setting for a particular workload. Workloads which seem promising are web server, scientific workloads [Miller91], and database benchmarking workloads.

Acknowledgments. Many thanks to Bruce Hillyer for many interesting conversations.

References

[Baker91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Asilomar (Pacific Grove), CA), pages 198–212. ACM Press, October 1991.

[Barve99] Rakesh Barve, Elizabeth Shriver, Phillip B. Gibbons, Bruce K. Hillyer, Yossi Matias, and Jeffrey Scott Vitter. Modeling and optimizing I/O throughput of multiple disks on a bus. *Proceedings of the 1999 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems* (Atlanta, GA), May 1999. Available at `http://www.bell-labs.com/~shriver/`.

[Bastian82] A. L. Bastian. Cached DASD performance prediction and validation. *Proceedings of 13th International Conference on Management and Performance Evaluation of Computer Systems* (San Diego, CA), pages 174–177, Mel Boksenbaum, George W. Dodson, Tom Moran, Connie Smith, and H. Pat Artis, editors, December 1982.

[Cao94] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation* (Seattle, WA), pages 165–178, November 1994.

[Cao95] Pei Cao, Edward W. Felten, Anna Karlin, and Kai Li. A study of integrated prefetching and caching strategies. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems* (Ottawa, Canada), pages 188–197, May 1995.

[Chen93] Shenze Chen and Don Towsley. The design and evaluation of RAID 5 and parity striping disk array architectures. *Journal of Parallel and Distributed Computing*, **17**(1–2):58–74, January–February 1993.

[Feiertag72] R. J. Feiertag and E. I. Organick. The Multics input/output system. *Proceedings of the Third Symposium on Operating Systems Principles* (Palo Alto, CA), pages 35–41, October 1972.

[Griffioen94] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. *Proceedings of the 1994 Summer USENIX Technical Conference* (Boston, MA), pages 197–207, June 1994.

[Hartman93] John Hartman and John Ousterhout. Letter to the editor. *Operating Systems Review*, **27**(1):7–9, January 1993.

[Hofri80] M. Hofri. Disk scheduling: FCFS vs. SSTF revisited. *Communications of the ACM*, **23**(11):645–653, November 1980.

[Hospodor95] Andy Hospodor. Mechanical access time calculation. *Advances in Information Storage Systems*, **6**:313–336, 1995.

[Kimbrel96] Tracy Kimbrel and Anna R. Karlin. Near-optimal parallel prefetching and caching. *Proceedings of the 37th Annual Symposium on Foundations of Computer Science* (Burlington, VT), pages 540–549, October 1996.

[Kotz95] David Kotz. Disk-directed I/O for an out-of-core computation. *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing* (Pentagon City, VA), pages 159–166, August 1995.

[Kroeger96] Thomas Kroeger. Predicting file system actions from reference patterns. Department of Computer Engineering, University of California, Santa Cruz, Santa Cruz, CA, December 1996. Master's thesis.

[Kuratti95] Anand Kuratti and William H. Sanders. Performance analysis of the RAID 5 disk array. *Proceedings of International Computer Performance and Dependability Symposium* (Erlangen, Germany), pages 236–245, April 1995.

[Lei97] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. *Proceedings of the USENIX 1997 Annual Technical Conference* (Anaheim, CA), January 1997.

[Madhyastha97a] Tara M. Madhyastha and Daniel A. Reed. Exploiting global input/output access pattern classification. *Proceedings of Supercomputing '97* (San Jose, CA), November 1997.

[Madhyastha97b] Tara M. Madhyastha and Daniel A. Reed. Input/output access pattern classification using hidden Markov models. *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS)* (San Jose, CA), pages 57–67. ACM Press, November 1997.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–197, August 1984.

[McKusick96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley Publishing, 1996.

[Merchant96] Arif Merchant and Philip S. Yu. Analytic modeling of clustered RAID with mapping based on nearly random permutation. *IEEE Transactions on Computers*, **45**(3):367–373, March 1996.

[Miller91] Ethan L. Miller and Randy H. Katz. Input/output behavior of supercomputing applications. *Proceedings of Supercomputing '91* (Albuquerque, NM), pages 567–576, November 1991.

[Mowry96] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation* (Seattle, WA), pages 3–17. USENIX Association, October 1996.

[Ng91] Spencer W. Ng. Improving disk performance via latency reduction. *IEEE Transactions on Computers*, **40**(1):22–30, January 1991.

[Patterson95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, CO), pages 79–95. ACM Press, December 1995.

[Ritchie78] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, **57**(6):1905–1930, July/August 1978. Part 2.

[Ruemmler94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, **27**(3):17–28, March 1994.

[Seeger96] B. Seeger. An analysis of schedules for performing multi-page requests. *Information Systems*, **21**(5):387–407, July 1996.

[Seltzer90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. *Proceedings of Winter 1990 USENIX Conference* (Washington, DC), pages 313–323, January 1990.

[Shriver97] Elizabeth Shriver. *Performance modeling for realistic storage devices.* PhD thesis. Department of Computer Science, New York University, New York, NY, May 1997. Available at `http://www.bell-labs.com/~shriver/`.

[Shriver98] Elizabeth Shriver, Arif Merchant, and John Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. *Joint International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '98/Performance '98)* (Madison, WI), pages 182–191, June 1998. Available at `http://www.bell-labs.com/~shriver/`.

[Small98] Christopher Small. *Building an extensible operating system.* PhD thesis. Division of Engineering and Applied Sciences, Harvard University, Boston, MA, October 1998.

[Wilhelm77] Neil C. Wilhelm. A general model for the performance of disk systems. *Journal of the ACM*, **24**(1):14–31, January 1977.

[Worthington94] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA), pages 241–251, May 1994.

A Validation of the model

To validate the model, we ran a set of simple microbenchmark workloads, collecting traces of both file system and disk events. From these traces, we determined the mean disk and application-level response times for a variety of synthetic and real workloads. The results in this section are presented for the machine with the slower bus.

The workloads. We created simple workloads which opened files, read them sequentially, and closed them. We recorded only the reads, i.e., not the open and close operations. We varied the request size and size of the file (which, in turn, varied data_span and run_length). Our workloads have a closed arrival process; files greater than 96 KB spanned at least two cylinder groups.

Single files accessed. We ran our workloads using one process to access a single file; we repeated the experiments 100 times and averaged to compute the measured FSRT. Table 3 includes the measured and model-computed FSRT using equations (4) and (5), as well as the relative errors of the model. In all but one case the model's estimate is within 4% of the measured result; when reading 32 KB of a 64 KB file the model error is 9%, which is higher, but still quite good for an analytic model of this type.

Multiple files accessed. We then reran our experiments with multiple concurrent processes, each accessing a different file with the same workload specification (i.e., same request size and same file size). All of the files were opened at the beginning of the experiment; the files which were used during one experiment run were randomly chosen among a set of files that spanned the 4 GB disk. We flushed the disk cache between each of the 50 experiment runs. Table 4 presents our results using equations (4) and (5). We see that the model's error is 6% or less in all cases.

Prefetching disabled. We modified the file system to disable prefetching and reran our single file experiments. Table 5 contains our results using equations (4) and (5) with E[disk_request_size] = BlockSize. The maximum relative error is 7%.

Table 3: Relative errors with one file accessed.

request size (KB)	*file size* (KB)	*measured* **E**[FSRT] (ms)	*computed* **E**[FSRT] (ms)	*error*
8	64	2.16	2.16	0%
16		4.15	4.31	4%
32		7.88	8.62	9%
64		17.41	17.24	1%
8	128	2.70	2.69	1%
16		5.41	5.37	1%
32		10.74	10.74	0%
64		21.08	21.48	2%

Table 4: Relative errors with multiple files accessed.

number of files	*request size* (KB)	*file size* (KB)	*measured* **E**[FSRT] (ms)	*computed* **E**[FSRT] (ms)	*error*
2	8	64	8.80	8.80	0%
	16		17.27	17.61	2%
	32		34.89	35.22	1%
	64		68.96	70.43	2%
3	8	64	12.82	12.90	1%
	16		25.60	25.81	1%
	32		50.85	51.61	1%
	64		99.98	103.23	3%
4	8	64	24.61	23.09	6%
	16		49.16	46.17	6%
	32		98.10	92.35	6%
	64		193.86	184.70	5%
8	8	64	43.83	43.52	1%
	16		87.54	87.04	1%
	32		174.06	174.08	0%
	64		344.04	348.16	1%
16	8	64	77.96	79.50	2%
	16		155.48	159.00	2%
	32		308.20	318.01	3%
	64		604.90	636.01	5%

Table 5: Relative errors with one file accessed and prefetching disabled.

request size (KB)	*file size* (KB)	*measured* **E**[FSRT] (ms)	*computed* **E**[FSRT] (ms)	*error*
8	64	2.64	2.46	7%
16	64	5.22	4.93	6%
32	64	10.21	9.86	3%
64	64	19.80	19.72	0%
8	128	3.26	3.07	6%
16	128	6.59	6.14	7%
32	128	13.21	12.28	7%
64	128	26.19	24.56	6%

The Region Trap Library: Handling Traps on Application-Defined Regions of Memory

Tim Brecht
Department of Computer Science
University of Waterloo, Waterloo, ON Canada
brecht@cs.uwaterloo.ca

Harjinder Sandhu
Department of Computer Science
York University, Toronto, ON Canada
hsandhu@cs.yorku.ca

Abstract

User-level virtual memory (VM) primitives are used in many different application domains including distributed shared memory, persistent objects, garbage collection, and checkpointing. Unfortunately, VM primitives only allow traps to be handled at the granularity of fixed-sized pages defined by the operating system and architecture. In many cases, this results in a size mismatch between pages and application-defined objects that can lead to a significant loss in performance. In this paper we describe the design and implementation of a library that provides, at the granularity of application-defined regions, the same set of services that are commonly available at a page-granularity using VM primitives. Applications that employ the interface of this library, called the Region Trap Library (RTL), can create and use multiple objects with different levels of protection (i.e., invalid, read-only, or read-write) that reside on the same virtual memory page and trap only on read/write references to objects in an invalid state or write references to objects in a read-only state. All other references to these objects proceed at hardware speeds.

Benchmarks of an implementation on five different OS/architecture combinations are presented along with a case study using region trapping within a distributed shared memory (DSM) system, to implement a region-based version of the lazy release consistency (LRC) coherence protocol. Together, the benchmark results and the DSM case study suggest that region trapping mechanisms provide a feasible region-granularity alternative for application domains that commonly rely on page-based virtual memory primitives.

1 Introduction

Modern operating systems typically export to the user-level the ability to manipulate the protection levels of virtual memory pages and to handle traps to those pages from within an application. Although originally intended for user-level virtual memory management, these mechanisms have been used in many application domains beyond those for which they were originally designed, including distributed shared memory, persistent stores, garbage collection and checkpointing. Unfortunately, pages are often the wrong unit for data management, since their size is fixed by the operating system and architecture and this size usually has little in common with the size of variable-length data objects defined within an application. When multiple data objects with different access patterns occupy the same virtual memory page, a trap to one object on the page may adversely affect the state of any other object on the same page. Conversely, for data objects that cross page-boundaries, traps must typically be handled one page at a time and may incur greater overhead than necessary.

A variety of mechanisms have been proposed for managing data at finer granularity or at a granularity defined by the application. There has been much research, for example, on the use of software checks, inserted by the compiler prior to memory references, to determine the status of persistent objects in a persistent store (e.g., White [26] and Moss [18]). Some systems have also used software checks to implement fine-grained sharing or write collection in a distributed shared memory (DSM) system e.g., Shasta [22], Blizzard-S [23], and Midway [6]). Although the trade-offs between incurring a small software check overhead on common memory references versus a large overhead on traps (which are much less common) have been studied from an efficiency point of view (Hosking and Moss [10], Zekauskas *et al.* [28], Thekkath and Levy [25]), one of the advantages of VM trap handling mechanisms seems to be their availability on most modern operating systems and architectures. Other object-based DSM systems have explored the use of program-level constructs to explicitly inform the system when shared objects are referenced (e.g., Orca [5], Amber [9], Midway [6] and CRL [13]). Although these systems avoid the need for both software checks and traps, they also impose a more restrictive programming model on the user.

In this paper, we present the design of mechanisms for providing, at the granularity of application-defined regions of memory, the same set of services that are commonly available at a page-granularity using virtual memory primitives. These mechanisms, implemented as a C library called the Region Trap Library (RTL), use a combination of pointer swizzling and virtual memory protection to provide a portable set of primitives for manipulating protection levels to regions and for handling traps to regions from within an application domain (and they do not depend on features of a particular programming language). Thus, for example, if three regions A, B, and C use the RTL interface and occupy the same virtual memory page, and A is in an invalid state, B is in a read-only state, and C is in a read-write state, the RTL mechanisms will generate a trap on a read or write reference to A or a write reference to B, but allow read/write references to C and read references to B to proceed at hardware speeds. The RTL mechanisms also allow an application domain to map the same region at different protection levels for different threads within the same address space, and to determine both the set of regions that have been modified since a previous check and the set of modified addresses within those regions. Together, these services form a superset of those commonly offered by the operating system through VM primitives at a page-granularity and listed by Appel and Li in their paper discussing the requirements of application domains that make use of VM primitives [4].

One of the main contributions of the mechanisms described in this paper is that they are general purpose (i.e., they can be used in different application domains and languages) and they can be ported to many different modern architectures and operating systems. While earlier papers have discussed the use of pointer swizzling for object faulting [10][25], to our knowledge pointer and register swizzling have been implemented previously only within an interpreted Smalltalk environment and specifically for persistent storage [10]. Earlier systems have also not considered the problem of providing more than one level of protection using these techniques, or the problem of providing a solution that can work across different architectures or in application domains other than persistent storage and garbage collection. We have currently implemented the RTL mechanisms on several operating system/architecture combinations: Solaris/MicroSparc, Solaris/UltraSparc, IRIX/R4400, AIX/PowerPC, and LINUX/Pentium, and explored their use within a distributed shared memory system. We present some of the issues involved in the design and implementation of these mechanisms on different systems and the overhead incurred by these mechanisms on each of these different systems. We also describe the implementation and use of these mechanisms for a region-based version of the *lazy release consistency* protocol within the TreadMarks DSM system. In the DSM case study, we find that the overhead estimates for region trapping account for less than 1% of the parallel execution time in five of the six applications examined, and 6% in the other application. The use of regions rather than pages for sharing data also leads to significant improvements (up to 41%) in performance for the applications used in this study.

An overview of this paper follows. Section 2 presents some of the background and related work for the ideas presented in this paper. Section 3 describes the design and implementation of the Region Trap Library. Section 4 presents the results of some micro-benchmarks that compare the overhead of region trapping mechanisms to page-based VM primitives. Section 5 presents our case study using region trapping within a DSM environment. In Section 6 we discuss the potential and limitations of our approach and present our conclusions in Section 7.

2 Background and Related Work

Our goal in this paper is to explore the design and implementation of mechanisms for modern architectures and modern operating systems that offer the same set of services typically provided through virtual memory primitives, but at the granularity of user-defined regions. Appel and Li, in their paper on virtual memory (VM) primitives for user applications, list the set of services commonly required by applications that make use of these VM primitives [4]. These services, generalized to include region-based primitives in addition to VM primitives, are shown in Table 1. These services are available in some form or another on virtually all modern architectures but only at a page-granularity. Some very early architectures (e.g., the Burroughs 6000 series of computers [16]) once provided non-page granularity support for handling traps and managing data from user applications, but such support is not available on any modern architectures.

The mechanisms we propose use pointer swizzling for region trap handling. Pointer swizzling has been used previously in persistent object and garbage collection systems. In a persistent object system, pointers to objects may have different representations when they reside on disk than when they reside in memory, and pointer swizzling is used to update the values of pointers when objects are brought into memory or written out to disk. Most implementations use either software checks to detect references to invalid objects (see White [26] for a review) or VM page trapping mechanisms to fault ob-

Primitive	Description
TRAP	handle traps to {page,region} in user handler
PROT1	decrease accessibility of a {page,region}
PROTN	decrease accessibility of N {pages,regions}
UNPROT	increase accessibility of a {page,region}
DIRTY	return the list of dirty {pages,regions}
MAP2	map physical {page,region} at two different addresses at different protection levels, in the same address space

Table 1: Description of services typically required of applications using page-based VM primitives, and analogous region-based RTL services.

jects into memory at a page-granularity (e.g., Texas [24] and Objectstore [15]).

Hosking and Moss [10] implemented a technique called *object faulting* within an interpreted Smalltalk environment. In their strategy, pointers to a persistent object are swizzled to refer to a fault block that lies on a protected page. This fault block acts as a stand in for the object when it is not in memory. A reference to the object generates a trap, and the object is faulted in to memory. In this scheme, references to the object once it is brought into memory are indirect, unless, as suggested by Hosking and Moss, a garbage collection system is used that can recognize these indirect references and convert them to direct references. Although this strategy provides some advantages over persistent object systems that use page-granularity trapping mechanisms or software checks prior to each memory reference, its implementation is language dependent and it makes extensive use of virtual method invocation and built-in indirect references to objects within an interpreted Smalltalk environment.

Thekkath *et al.* describe the use of *unaligned access traps* for object faulting [25]. Unaligned access traps are generated by some architectures on memory references to data that should to be word aligned but is not. This mechanism was used for fast synchronization in the APRIL processor [2]. Using this approach for object faulting, pointers to an object would be swizzled so that they are unaligned, and a subsequent dereference to the pointer would generate an unaligned access trap that could be handled by the application. However, this strategy, while providing a language independent solution to object granularity trap handling, cannot be used on architectures that do not support unaligned access traps (e.g., the PowerPC architecture), and will only work in limited cases on architectures that support unaligned traps for some memory reference instructions but byte level accesses for others (e.g., the SPARC architecture).

The mechanisms we describe in this paper are similar in some respects to these latter two strategies, but are designed to provide greater functionality using mechanisms that do not depend on features of a particular programming language, and in an architecturally portable fashion. Many applications that use page-based virtual memory primitives require the ability to trap on read/write references to inaccessible pages *or* on write references to read-only pages. However, the use of unaligned access traps, as well the strategy proposed by Hosking and Moss, provide only the ability to trap on inaccessible objects. One level of protection may be sufficient in the context in which these earlier strategies have been proposed (to fault objects into memory), but they do not suffice in many other contexts. This includes checkpointing applications, where the system has to be able to detect write operations to objects that are either in a read-only or invalid state, or in the implementation of coherence protocols within a distributed shared memory system, where the system typically needs to be able to obtain updates to an object on read references in an invalid state and to mark changes to the object on write references in the invalid or read-only state.

3 The Region Trap Library

In this section, we describe the design and implementation of the Region Trap Library (RTL). We begin by describing the interface to the RTL.

3.1 RTL Interface

Applications that use the RTL must identify the areas of memory that are to be managed as regions, the set of pointers that are used to reference regions, and a handler function that will be invoked when a region trap occurs. A variety of primitives are provided for specifying each. To identify regions, an application may use a memory allocator called `region_alloc(size)` to both allocate and define a region, or define a previously allocated range of memory as a region using `region_define(addr, size)`. An additional parameter may be provided to these functions that identifies a pointer that will be used to reference that region. A region pointer identified in this way is referred to as a *bounded* region pointer, since it can only be used to refer to the specified region. Additional bounded pointers to the same region can be specified using a primitive called `region_bptr(&ptr1, &ptr2)`, which indicates that `ptr2` will be used to refer to the same region as `ptr1`. *Unbounded* region pointers, those that may refer to any region, can be specified using a primitive called `region_ptr(&ptr)`. Region pointers can be destroyed using a call to `region_ptr_free(&ptr)`. The `region_ptr` and `region_ptr_free` calls are used to maintain a list of pointers associated with each

Page-based trapping using VM primitives	RTL region trapping with a bounded region pointer	RTL region trapping with an unbounded region pointer	RTL region trapping with C++ pointer declaration

char *x;
struct sigaction s1,s2;
...
x = valloc(N);
...
s1.sa_handler = page_handler;
sigaction(SIGSEGV,&s1,&s2);
mprotect(x,N,PROT_READ);
...
a = x[0]; /* no trap */
x[1] = 1; /* trap */
x[2] = 2; /* no trap */
	char *x;
	
	...
	x = region_alloc(N,&x);
	...
	
	region_handler(x,reg_handler);
	region_protect(x,PROT_READ);
	...
	a = x[0]; /* no trap */
	x[1] = 1; /* trap */
	x[2] = 2; /* no trap */
		char *x;
		
		...
		x = region_alloc(N);
		...
		region_ptr(&x);
		region_handler(x,reg_handler);
		region_protect(x,PROT_READ);
		...
		a = x[0]; /* no trap */
		x[1] = 1; /* trap */
		x[2] = 2; /* no trap */
			region_ptr<char *> x;
			
			...
			x = region_alloc(N);
			...
			
			region_handler(x,reg_handler);
			region_protect(x,PROT_READ);
			...
			a = x[0]; /* no trap */
			x[1] = 1; /* trap */
			x[2] = 2; /* no trap */

Table 2: Example of how regions are defined in the RTL using both the C and C++-style declaration of region pointers, and compared to the analogous code using page-based virtual memory mechanisms.

Trap handler for virtual memory primitives	Trap handler for region trapping library

page_handler(signal-context)
{
char *addr = faulting_address();
...
 	
if (is_a_write_fault)
mprotect(addr,page_size,PROT_READ | PROT_WRITE);
else
mprotect(addr,page_size,PROT_READ);
}
	reg_handler(region-trap-context)
	{
	char *addr = faulting_address();
	...
	
	if (is_a_write_fault)
	region_protect(addr,PROT_READ | PROT_WRITE);
	else
	region_protect(addr,PROT_READ);
	}

Table 3: Skeletons of trap handler functions contrasting the way traps are handled using virtual memory primitives and the region trapping mechanisms.

region. Later, when protection levels are changed for a region each of these pointers will be swizzled by the RTL (see Section 3.2 for further details).

All calls to allocate or define regions, or to modify protection levels, update the region pointers that are declared by the user to reference that particular region. Region pointers can be used in the same way as other C pointers including the ability to use offsets within a region and to manipulate pointers using pointer arithmetic. The only restriction is that pointers not explicitly identified to the RTL should not be used to reference regions allocated or defined by the RTL.

The region handler function, potentially a different one for each region, can be provided as an additional parameter when the region is defined, or it may be specified separately using a primitive called `region_handler`. Region protection levels are set using a function called `region_protect`. Protection levels are specified as they are for the VM primitive `mprotect`, as `PROT_NONE` (to trap on any subsequent reference to this region), `PROT_READ` (to trap only on subsequent write references to this region), and `PROT_READ | PROT_WRITE` (to enable all read or write references to this region).

These C functions present a low-level interface that is intended to mimic corresponding page-based VM primitives whenever possible. Table 2 shows how VM primitives (column 1) and RTL primitives using this C interface (columns 2 and 3) are used for handling traps, to virtual memory pages in the former case and for regions in the latter case. Table 3 shows the skeleton of analogous application trap handler functions for both the VM and region trapping cases. While the use of C as an interface permits this library to be used in virtually any language environment, the semantics of C are such that a programmer using this interface directly would have to exercise some discipline in the way regions and region pointers are defined and used. For instance, there is no way in C to indicate to the RTL when a pointer is declared that it will be used to refer to regions, and to automatically inform the RTL library that a pointer is no longer being used to refer to regions when the scope within which it was declared has ended. This necessitates the use of the routines `region_ptr` (and `region_ptr_free` for pointers that are not declared globally) when using the C interface directly.

Within specific languages and application domains, higher-level interfaces can be built that provide more elegant ways to define regions, region pointers, and trap handlers. In C++, for example, the template facility provides a means to simplify the way in which region pointers are allocated and deallocated from the RTL, through a region pointer wrapper class as shown in column 4 of Table 2. Using C++ templates, the declaration of a region pointer as `region_ptr<type *>` is sufficient both as a declaration of the pointer and an indication to the RTL that this pointer will be used to refer to regions. Calls to the RTL functions `region_ptr` and `region_ptr_free` are made automatically from within the constructor and destructor for this pointer class. The access operators for these wrapper classes are overloaded so that references to a pointer declared in this way are statically replaced by the compiler with direct pointer references.

3.2 Implementing Protection Levels

Within the RTL, three separate memory areas are maintained, an *invalid* area, a *read-only* area, and a *read-write* area. The invalid area does not occupy any physical memory and may in fact be outside of the address space of the process (e.g., in the kernel's address space). The other two areas are each composed of a set of virtual memory pages that are mapped into the user address space. In the read-only area, all pages are mapped read-only, and in the read-write area, all pages are mapped read-write. Space is allocated to a specific region in each of these areas on demand and according to the protection levels that are used for that region. When region protection for a particular region is set to x (where x is invalid, read-only, or read-write), pointers to that region are swizzled so that they refer to the copy of the region occupying pages in the area mapped to protection level x. This strategy is illustrated in Figure 1.

If a region is currently in an invalid state, pointers to that region point to an area within the invalid space reserved for that region (Figure 1(a)), and a reference to that region through any of these pointers will generate a trap. If a region is currently in a read-only state, pointers to that region refer to the copy of the region in the read-only area (Figure 1(b)), and only write references to that region will generate a trap. Finally, if a region is in a read/write state (Figure 1(c)), pointers to that region are swizzled to point to a space reserved for that region in the read/write area, so that all references to that region can proceed without a trap. In this way, a lower protection level for one region will not result in unnecessary traps to another region occupying the same set of virtual memory pages, since the pointers to each respective region will simply point to different areas. Mapping the same region at two different protection levels within the same address space can be done by declaring two different sets of pointers to the same region but with different protection levels.

3.3 Region Trap Handling

When a region trap occurs, the RTL goes through the following sequence of steps:

- Determine the region to which the faulting address belongs, the region pointer(s) that refer to this region, and the application's trap handler for that region.

- If the protection level is being changed within the region trap handler, decode the faulting instruction and then swizzle both the register containing the faulting address and pointer(s) in memory referring to the faulting region. Recall that the pointers referring to each region have been identified using `region_ptr` or `region_bptr` calls.

Our prototype implementation of the RTL uses a binary search to locate region addresses stored within an AVL tree. Although faster implementations of region lookup are possible, this approach provides adequate lookup times for the applications on which we have conducted experiments. Once the region is located, the trap handler specified by the application for that region is invoked. Region pointers bound to this region and the address of the application's handler function for this region are stored within the nodes of this tree and thus require no additional effort to locate. Unbounded region pointers, if any have been declared, must be searched separately to see if any refer to this region. It is the responsibility of the application's trap handler to determine how to handle a region trap and to indicate to the RTL what state the region should be mapped to prior to returning from the handler (as in Table 3).

When `region_protect` is invoked from the application's handler, the register and all pointers known to be referring to the region (as specified as parameters to `region_alloc`, `region_define`, `region_ptr`, or `region_bptr`) are swizzled. Swizzling the register containing the faulting address requires first decoding the instruction that generated the fault in order to determine which register requires swizzling, and then modifying that register. Region pointers may point to any address within a region. Consequently, when pointers or registers are swizzled from one memory area to another, their offset from the start of the region in those respective areas must be preserved (this is what permits offsets and pointer arithmetic to be used).

A key concern with the efficiency of this strategy arises from the fact that some transitions to the read-only state for a region require *updating* the read-only copy of

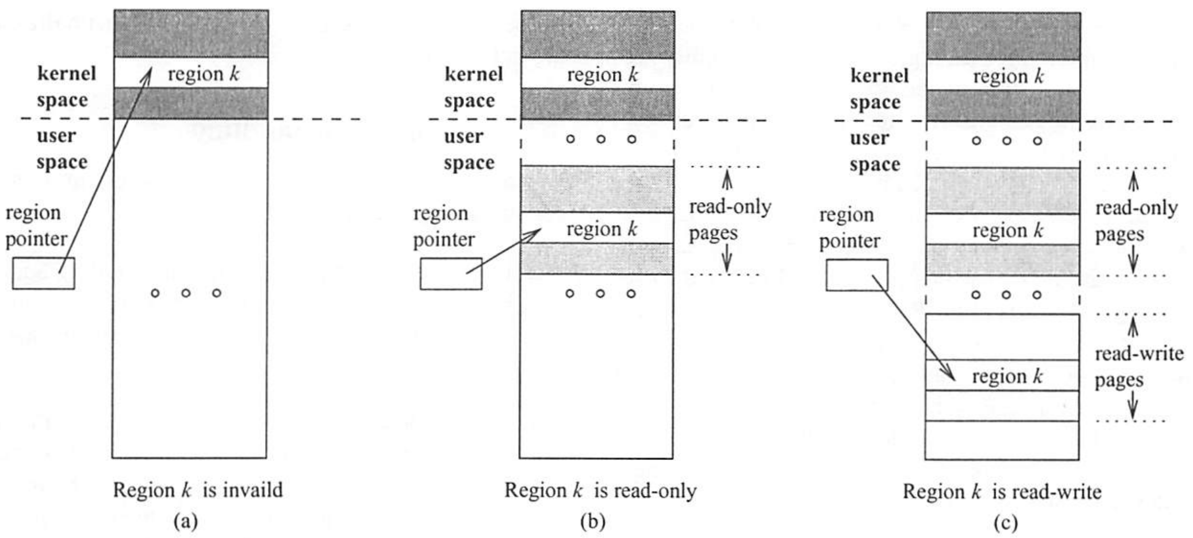


Figure 1: Setting region protection levels by swizzling region pointers between three areas: invalid, read-only, and read-write. On architectures with a large kernel address space (e.g., MIPS), the invalid area is mapped to kernel space. On other architectures, the invalid area is mapped to user space but does not occupy physical memory.

the region, in order to maintain consistency. In particular, on transitions directly from the read-write state to the read-only state, or transitions from the read-write state to the invalid state and then to the read-only state, this update must be performed. Further, the pages occupied by the read-only copy of the region are themselves in a read-only state, and they must first be unprotected (using `mprotect`) prior to beginning the update, and then reprotected once the update is complete. The performance implications of this read-only copy update are considered in the benchmarks of Section 4. Regions that transition only between read/write and invalid states or read-only and invalid states do not incur this overhead.

The allocation of the region in two areas of memory, a read-only area and a read-write area, facilitates an additional service that many applications can make use of which is often referred to as a *diff*. Using this service, an application may, at any point during execution, query the RTL to determine the set of memory locations in a region that have been modified since a previous write trap or checkpoint operation. Many distributed shared memory systems and checkpointing applications, for example, implement such a service at a page-granularity. The region-based distributed shared memory implementation within Treadmarks, described in Section 5 also exploits this service, replacing Treadmarks' existing page-based diff mechanisms with region-based diff mechanisms.

3.4 Implementation Issues

The RTL has currently been implemented on the MicroSparc and UltraSparc architectures, both running Solaris, the SGI MIPS R4400 architecture running IRIX, the IBM RS/6000 architecture running AIX, and the Pentium architecture running LINUX. Some of the issues involved in implementing these strategies on various architectures, and in particular, some of the requirements from the operating system, architecture, and compiler for an implementation of region trap handling on these and other platforms are discussed in this subsection.

Architectural requirements

Setting region protection levels to invalid requires the ability to swizzle pointers to an area that is guaranteed to generate a trap. The MIPS architecture, one of the four architectures on which we have currently implemented the RTL, makes this particularly easy. On the MIPS, addresses with the high order bit set refer to kernel space addresses, and the region trapping implementation can take advantage of this by swizzling this high order bit on pointers to regions in an invalid state. On the other architectures, our implementation creates an additional area within a processes' address space that is mapped as inaccessible and never occupies physical memory. Invalid regions are mapped to this area. These two approaches for mapping the invalid area behave the same and use the same amount of physical memory, but the latter solution potentially occupies three areas in the virtual memory address space of a process for each region.

The use of precise interrupts on protection violations by the architecture is an important prerequisite for the implementation of region trap handling. In the absence of precise interrupts, the trap handler would have a more difficult time determining which instruction/register caused the fault, and what machine state may have been altered. Fortunately, most modern architectures (including all of the architectures discussed in this paper) use precise interrupts on protection violations.

Operating system requirements

The requirements from the operating system for an implementation of region trapping are relatively few. When a trap occurs and the RTL's trap handler is called, the operating system must provide enough processor context to the RTL trap handler to allow it to determine the faulting address, the faulting instruction, and the register that contains the faulting address. Additionally, trap recovery in the RTL requires the ability to modify and restore execution context. The versions of UNIX on which we have currently implemented the RTL (IRIX, AIX, LINUX, and Solaris) all provide this level of support.

Compiler requirements

There is some concern that the compiler might cache region pointers in registers and that these registers, which are not known to the RTL will not be swizzled upon subsequent changes in protection levels. As a result later references to the region using the cached register value will not generate the proper behaviour (either generating unwanted traps or not generating desired traps). Since protection levels are changed by calling the function `region_protect` and since the compiler should not cache values that can be changed inside of the function call across such calls, the compiler should not create these potentially dangerous register caches. The problem is exacerbated because protection levels might be changed by a function that is called asynchronously (e.g., as a result of a trap). In the benchmarks that we have run on the five different platforms, as well as in the distributed shared memory experiments we have conducted using the RTL in the SGI MIPS environment, we have not encountered any instances of this type of pointer aliasing. It is conceivable that some compilers may perform such aliasing within registers. Fortunately, if such a problem were to arise, by declaring region pointers as `volatile`, the compiler is forced to generate code that reloads a region pointer (albeit probably from the cache) each time it is dereferenced, and the compiler is then not able to create register aliases to regions. This solution is clearly more restrictive than necessary, and will result in some performance degradation. Ideally, compilers should also support a flag that prohibits the aliasing of pointers in registers without requiring these pointers to be declared as volatile.

4 Micro-benchmarks

In this section, we present the results of benchmarks that measure the overhead of region trapping mechanisms relative to standard VM page trapping primitives. These benchmarks are not designed to provide insight into overall application performance using page or region-based trapping mechanisms, since the two techniques will likely result in a different number of traps being generated. Instead, these benchmarks provide some indication of how well different architectures and operating systems support region trapping mechanisms and whether the overheads incurred by the region trapping mechanisms would be prohibitive for application domains that commonly make use of VM primitives. Section 4.1 describes the benchmarks that we use and Section 4.2 discusses the results of these benchmarks.

4.1 Benchmark Descriptions

Three benchmarks are used. The first, referred to as `trap`, measures the overhead of using trap mechanisms. For VM primitives, this is simply the cost of entering and exiting a signal handler. For RT primitives, the `trap` time also includes the additional cost (within the signal handler) of decoding the instruction, locating the region, calling a null application trap handler, and swizzling the region pointer and a register.

Appel and Li observe in their paper that applications that use virtual memory primitives typically perform one of the following two sequence of operations:

1. `Prot1`: protect one page and, on a subsequent trap to that page, unprotect the page from inside the trap handler, or

2. `ProtN`: protect a set of N pages and, on a trap to any of these protected pages, unprotect the page that caused the trap from inside a trap handler.

A comparison of these two sequences provides a better understanding of the relative overheads involved in using trapping mechanisms than simply measuring trap costs alone. Consequently, we use two benchmarks that are patterned after these two sequences. These two benchmarks, referred to as `Prot1` and `ProtN`, are constructed in the same way as described in the Appel and Li paper. In the `Prot1` test, a protected page or region is referenced and, inside the trap handler, the page or region is unprotected and another one is protected. In the `ProtN` test, 100 pages or regions are protected, and each one is referenced and unprotected one at a time within a trap handler.

A large number of repetitions of these sequences are conducted in order to obtain an average cost per sequence

OS	Arch	page	VM			RT-basic			RT-update		
		size	trap	prot1	protN	trap	prot1	protN	trap	prot1	protN
IRIX	MIPS R4400	4 KB	65	275	128	96	130	103	477	664	580
6.2	175 MHz					1.5	0.5	0.8	7.3	2.4	4.5
Solaris	MicroSparc	8 KB	286	778	591	352	365	366	2258	2539	2518
2.5.1	70 MHz					1.2	0.5	0.6	7.9	3.3	4.3
Solaris	UltraSparc	8 KB	94	200	167	112	119	127	854	910	830
2.5.1	168 MHz					1.2	0.6	0.8	9.1	4.5	5.0
LINUX	Pentium Pro	4 KB	12	39	30	23	32	28	115	198	174
2.0.0	200 MHz					1.9	0.8	0.9	9.6	5.1	5.8
AIX	PowerPC	4 KB	61	169	146	92	107	104	190	774	688
4.1	133 MHz					1.5	0.6	0.7	3.1	4.6	4.7

Table 4: Times (in microseconds) comparing VM primitives, RT-basic and RT-update. RT-basic refers to region traps on which the read-only copy of the region does not need to be updated, while RT-update refers to region traps on which the read-only copy of the region must be updated. RT region size is equal to the system page size. Numbers on the 2nd line for each system are the ratio of RT test costs relative to the corresponding VM test. Times reported are the average of multiple iterations of each test on different pages or regions.

(typically 10,000 to 100,000 repetitions were used, depending on the time taken to execute each sequence). However, because of the caching effects that occur as a result of doing multiple repetitions, all results should be considered optimistic. Additionally, the results do not consider overheads the RTL would incur as a result of swizzling multiple pointers (since the number of such pointers will typically be small and the overhead required to simply modify a pointer will be negligible) and the design of the `trap` benchmark does not consider overheads incurred to search for the faulting region. However, since the `Prot1` and `ProtN` benchmarks use 100 regions, the costs to find the appropriate region are included in those benchmarks. Although the time required to find the appropriate region in the RTL depends on the number of regions, similar overheads would be incurred using VM primitives if the action taken on a trap depends on the object generating the trap.

For all three benchmarks (`trap`, `Prot1`, and `ProtN`), region trap overheads are classified as RT-basic and RT-update. RT-basic measures the cost of region traps that *do not* require an update of the read-only copy of the region, while RT-update measures the cost of region traps that *do* require such an update. Region size plays a significant role in the cost of RT-update traps, but no role in the cost of RT-basic traps. The first set of tests use a region size that is equivalent to the size of a page on each system (8 KB on Solaris and 4 KB on the others). Subsequent tests show the effect of varying the region size on the `Prot1` benchmark.

4.2 Benchmark Results

Table 4 shows the results of the three tests on each of the systems on which our RTL prototype has been implemented. For each OS/architecture shown in this table, the second line shows the ratio of the cost of the RT benchmark relative to the equivalent VM benchmark. Thus, for example, RT-basic *trap* time under IRIX is 96 microseconds, while the VM *trap* time is 65 microseconds, resulting in a relative cost ratio of 1.5. A number less than 1 implies that the RT benchmark is faster than the equivalent VM benchmark.

Entering and exiting a signal handler requires crossing OS protection boundaries. This is fairly expensive on all architectures that we study, though the fast exception handling technique described by Thekkath and Levy [25] would significantly reduce this cost for both VM and RT primitives. Protection trap times on the LINUX/Pentium configuration are particularly low when compared with the other OS/architecture configurations, with trap times five times lower (12 μs) than the next best time (61 μs on the AIX system).

RT-basic trap costs are between 1.2 and 1.5 times more expensive than VM trap costs on all architectures except LINUX (due to its fast protection traps and more complex instruction set), where RT trap costs are 1.9 times more expensive. RT-update trap costs, as would be expected, are significantly higher, ranging from 3.1 times more expensive than VM trap costs on the AIX system to 9.6 times more expensive on LINUX. However, in our measurements of the components of this overhead, we found that for regions equivalent to or smaller than the size of a page, the cost of the `memcpy` function used to copy regions from the read-write area to the read-only area makes up only a small portion of this cost. Most of the additional cost for RT-update comes from the fact that the pages occupied by the read-only copy must be unprotected using `mprotect` before the update can begin, and then reprotected (again using `mprotect`) after

OS	Arch	region = 1 page		64 KB Regions			256 KB Regions		
		(VM)	RT-update	K pages	(VM)$*K$	RT-update	K pages	(VM)$*K$	RT-update
IRIX	MIPS R4400	275	664	16	4400	1548	64	17600	5650
6.2	175 MHz		2.4			0.4			0.3
Solaris	MicroSparc	778	2539	8	6224	8990	32	24896	23042
2.5.1	70 MHz		3.3			1.4			0.9
Solaris	UltraSparc	200	910	8	1600	1448	32	6400	3552
2.5.1	168 MHz		4.5			0.9			0.6
LINUX	Pentium Pro	39	198	16	624	464	64	2496	2606
2.0.0	200 MHz		5.1			0.7			1.0
AIX	PowerPC	169	774	16	2704	1915	64	10816	10151
4.1	133 MHz		4.6			0.7			0.9

Table 5: `Prot1` benchmarks for varying region size. K is the number of pages occupied by a region. All times shown are in microseconds. Times reported for RT-update are the average of multiple iterations of the `Prot1` test on the *same* region.

the update has completed. `mprotect`, a system call that requires crossing OS protection boundaries and shooting down TLB entries, is relatively expensive on all of the platforms we have used in conducting our experiments.

The `Prot1` and `ProtN` benchmarks reveal a very different picture than trap overheads alone. For the VM case, protection levels are set using `mprotect`. A `region_protect` in RT-basic is very cheap by comparison (typically less than 1 us), since it does not require any system calls. As a result, RT-basic `Prot1` times are smaller than VM `Prot1` times across all architectures, with differences ranging from a factor of 0.5 to 0.8. `Prot1` and `ProtN` times are almost identical for RT-basic, due to the small overhead of a `region_protect` call. For the VM `ProtN` test, protecting multiple pages in one call to `mprotect` is cheaper than protecting each page one at a time, so VM `ProtN` times are less than `Prot1` times. However, RT-basic `ProtN` times are still lower than VM `ProtN` times, by factors of 0.6 to 0.9.

RT-update `Prot1` times are significantly higher than RT-basic `Prot1` times, as would be expected. In RT-update however, since a significant proportion of the cost is due to using `mprotect` twice to unprotect and reprotect the page occupied by the region, and the VM `Prot1` test performs the same number of `mprotect` calls, RT-update `Prot1` times range from being just 2.4 times higher than VM `Prot1` on IRIX to 5.1 times higher on LINUX. RT-update `ProtN` times range from being 4.3 to 5.8 times higher than VM `ProtN` times.

4.3 Large Regions

In this subsection, we consider the effects of defining large regions that span multiple pages. Since some region traps require copying the region from one area to another, it stands to reason that region trapping overheads in these cases will increase significantly as the region size is increased, although the cost of using `mprotect` within a `region_protect` call in such cases will be amortized over the larger regions. The cost of handling traps to large regions is considered from two perspectives: (1) relative to the cost of handling a trap to a region for which the size is equivalent to one page, and (2) relative to the cost of handling a trap to a page-based strategy that would handle traps to the same number of pages as spanned by the region. For page-based strategies, the premise for the Appel and Li benchmarks is that typical applications incur a fault on each page separately regardless of how large the object is, since such strategies do not usually take application characteristics into account. Consequently, if a region spans K virtual memory pages in a region-based approach, the analogous page-based applications using VM mechanisms will likely incur K traps for every one trap incurred by the region-based approach.

Table 5 shows the results of the `Prot1` benchmark for region sizes equivalent to one 1 page, 64 KB and 256 KB. RT-basic times, which are not affected by region size, are not shown. Although experiments in the previous subsection were performed using multiple iterations of the same test across one hundred regions, this was not possible for the experiments here because of the large region sizes involved (declaring one hundred such regions results in paging on some of these machines). These tests are conducted using multiple iterations on the same region, and are thus more prone to caching effects than the experiments in the previous subsection. For comparison to the region trapping version, VM times show the cost of K `Prot1` sequences when the region spans K pages.

For 64 KB regions, although `memcpy` costs go up significantly, RT-update `Prot1` times increase by relatively small factors in the range of 1.6 (on the UltraSparc) to 3.5 (on the MicroSparc), when compared to the times for region sizes equivalent to the page size. RT-update times for the `Prot1` benchmark are lower than K `Prot1` sequences in the VM case for four of the systems studied (ranging from 0.4 to 0.9), and somewhat higher on the other (1.4 on the Solaris/MicroSparc system). This difference is reduced on some architectures and increased on others for 256 KB regions, so that the costs of RT-update relative to K VM `Prot1` tests range from 0.3 to 1.0. However, these latter comparisons assume for the VM case that every page spanned by the region would be referenced and that the VM strategy does not employ any strategy to increase the effective page size. A worst case comparison for the region trapping case would be in instances where only one of the K pages spanned by the region is actually referenced. In such cases, RT-basic costs, which are unaffected by region size, are the same relative to the VM case. However, RT-update costs would look significantly worse, by factors of 7 to 11 for 64 KB regions, when compared to a VM based strategy that handles a trap only to the page that was referenced.

4.4 Benchmark Summary

The benchmark results of this section provide some insight into the overheads involved in the use of the region trapping mechanisms described in this paper. The architectures on which region trapping mechanisms were implemented and studied vary significantly in speed and in the complexity of the instruction sets. Operating system overheads also play a significant role in these costs. Overall, despite the seemingly high overhead of keeping the read-only copy up-to-date with respect to the read-write copy, region trapping overheads are competitive with VM overheads. For instance, for regions equal to the size of a page, the `Prot1` benchmarks show region trapping to be faster by factors of 0.5 to 0.8 for transitions on which the read-only copy is not updated, and slower by factors of 2.4 to 5.1 when the read-only copy does need to be updated. For much larger regions, RTL costs in the `Prot1` benchmark are typically comparable to or lower than VM costs when K traps to a page in the VM case are considered equivalent to one trap in the RTL case for regions spanning K pages.

Since these benchmarks provide only a microscopic view of RTL and VM overheads, the question of how real applications will perform using these mechanisms cannot be answered without examining the applications themselves. In particular, the number of traps that are actually generated is entirely application dependent and is likely to be different within a page-based and region-based version of the same application. For instance, when multiple objects occupy a single page, a page-based strategy may generate more traps than a region-based strategy (as a result of false sharing for example), or fewer (if all of the objects on that page are accessed together). Conversely, if an object is much larger than a page, a page-based strategy may generate more traps than a region-based strategy if it faults on each page of the object separately, or an equivalent and perhaps fewer number of traps if not all of the pages occupied by an object are typically referenced at one time.

To obtain a clearer picture of how RTL mechanisms would behave within a real application domain, we implemented a region trapping based coherence protocol within a distributed shared memory system. This is described in the next section.

5 Case Study: Distributed Shared Memory

This section presents some results from a case study which uses region trapping within the TreadMarks distributed shared memory (DSM) system. TreadMarks uses page-based VM primitives to implement an efficient coherence protocol called *lazy release consistency* (LRC), described in detail by Keleher *et al.* [14]. We have modified TreadMarks to support a region-based version of LRC that uses region trapping to handle traps and manage data at the granularity of regions rather than pages.

For comparison, we have designed and implemented another coherence protocol, called Multiple-Writer Entry Consistency (or MEC) which is described in detail in an earlier publication [20]. This protocol is similar to *entry consistency* [6] from the programming perspective except that it uses program-level annotations that are non-synchronizing. In this paper versions of the applications that have been implemented using this protocol are referred to as the Annotated Regions (AN) versions. They behave like the region trapping versions except that they use program-level annotations rather than traps to indicate when regions are referenced for read or write.

While the annotated (AN) versions are significantly more difficult to program, a comparison between the region trapping and annotated regions versions of these applications highlights the overhead of the region trapping mechanisms. At the same time, a comparison between the page-based version and region-trapping versions highlights both the cost of using region trapping versus VM mechanisms as well as the trade-offs between using regions rather than pages for data management.

Six applications were used in this study: matrix multiplication (MM), red-black successive over-relaxation (SOR), blocked contiguous LU-decomposition (LU), a Floyd-Warshall algorithm for finding shortest paths in a directed graph (FLOYD), integer sort (IS), and the traveling salesperson problem (TSP). TSP, SOR, and IS are all from the suite of applications used in earlier TreadMarks studies [1], LU is from the Splash-2 benchmark

Treadmarks version (VM)	Region Trapping version (RT)	Annotated Regions version (AN)
float **red, **black	float **red, **black	float **red, **black
		int *redX, *blackX
. . .	. . .	. . .
for (i=0;i<M+1;i++) {	for (i=0;i<M+1;i++) {	for (i=0;i<M+1;i++) {
red[i] = Tmk_malloc(NS)	red[i] = region_alloc(NS, &red[i])	red[i] = region_alloc(NS,&redX[i])
black[i] = Tmk_malloc(NS)	black[i] = region_alloc(NS, &black[i])	black[i] = region_alloc(NS,&blackX[i])
}	}	}
. . .	. . .	. . .
for (j=begin;j<=end;j++) {	for (j=begin;j<=end;j++) {	for (j=begin;j<=end;j++) {
for (k=0;k<N;k++) {	for (k=0;k<N;k++) {	writeaccess(blackX[j])
black[j][k] = (red[j-1][k] +	black[j][k] = (red[j-1][k]+	readaccess(redX[j-1])
red[j+1][k] + red[j][k] +	red[j+1][k] + red[j][k] +	readaccess(redX[j+1])
red[j][k+1])/4.0;	red[j][k+1])/4.0;	readaccess(readX[j])
}	}	for (k=0;k<N;k++) {
}	}	black[j][k] = (red[j-1][k] +
		red[j+1][k] + red[j][k] +
		red[j][k+1])/4.0;
		}
		}

Table 6: Snapshots of some code within SOR using VM, RT, and AN. M is the number of rows and NS is the size of each row.

suite [27], and MM and FLOYD were written locally.

5.1 Programming with Regions

Table 6 contrasts how one of the applications used in our study, SOR, is written to use each of the three protocols that we compare, page-based LRC (VM), region trapping LRC (RT), and annotated regions LRC (AN). Only a portion of SOR is shown but the example illustrates the program-level differences between these three approaches. In the original TreadMarks system, shared data must be allocated dynamically using the `Tmk_malloc` routine. In the original VM version of SOR (obtained with the TreadMarks distribution), each row of the two matrices (called `red` and `black`) is allocated separately using `Tmk_malloc`. In the RT version, each row is defined as a region by changing the `Tmk_malloc` call to `region_alloc` and providing a pointer to that region as a parameter. The rest of the code is identical for both page-based and region trapping versions of SOR. In the annotated regions version, each region is explicitly associated with a region identifier for that region (`redX` and `blackX` in the example), that is used in subsequent `readaccess` or `writeaccess` calls to identify a series of references to the region. Obviously a key motivation for implementing the RTL is to avoid having to annotate programs as shown in the AN example.

The other five applications required similar modifications to implement region trapping and annotated versions, although some of these applications use aliases to region pointers that also need to be declared as region pointers. In MM, all of the rows in a matrix used by a single processor are aggregated into a single region. In FLOYD, each row of the shared matrix is defined as a single shared region. In LU, each block is laid out contiguously and allocated separately in the original Splash-2 version. These blocks are allocated as regions in the RT version. In IS, there is a single shared data structure, a shared bucket, which is defined as a region. Finally, the original TreadMarks version of TSP allocates a single block of shared data using a single `Tmk_malloc` call. This block contains several different data structures. The RT version separates some of these data structures into separate regions, the largest of which is still about 700 KB in size.

5.2 Performance and RTL Overhead

We have conducted a series of experiments on a cluster of four 175 MHz R4400 SGI workstations connected by 155 Mbps links to a Fore Systems ASX-200 ATM switch. Table 7 shows the problem sizes used in these experiments, the size and number of regions defined in the region trap and annotated region-based versions (only the main regions are described), and the execution times of the applications on one processor and on four processors for each of the three models. Table 8 shows the number of traps that occur on a typical processor in the RT and VM versions of the applications, the RT overhead as a percentage of the runtime, and the number of messages and bytes transmitted between processors (relative to one processor) for both the RT and VM versions.

app	problem size and shared data structure	regions		execution times			
		num	size	1	VM	RT	AN
MM	640x640 matrices	9	$\approx$ 1 MB	81	33	22	22
SOR	200x4096 matrices, 100 iterations	4097	8 KB	46	27	16	15
LU	1024x1024 matrix, blocksize = 64	1024	32 KB	59	34	21	22
FLOYD	567 node graph	567	2.2 KB	137	64	48	48
TSP	19 city tour vector	1	1 MB	87	73	60	51
IS	2^{22} keys, bucketsize = 2^9, 10 iterations	1	2 KB	22	14	10	10

Table 7: Applications used in DSM study and the corresponding problem size descriptions and execution times (in seconds) on one processor and under VM, RT and AN on four processors.

app	Traps (#)		RT overhead	Messages		KBytes	
	RT	VM	% of runtime	RT	VM	RT	VM
MM	3	600	< 0.1%	7	606	2458	2463
SOR	548	2545	1%	490	1863	3226	3256
LU	201	1327	0.5%	528	2068	11475	6495
FLOYD	709	1702	< 0.1%	1721	2559	119	2054
TSP	309	2045	0.3%	940	3320	1077	711
IS	40	960	6%	93	1013	4021	3914

Table 8: Total number of traps generated, and messages and kilobytes transferred for RT and VM and the estimated RT overhead as a percentage of parallel execution time.

Region trapping overheads are estimated by multiplying the number of traps incurred of each type by the cost measured for that type of trap on the appropriate architecture (as shown in Table 4). These overheads account for less than 1% of the parallel execution time in five of the six applications, and 6% in the other application (TSP). Those overheads that do exist arise largely from the cost of updating the read-only copy of the region (when required) on transitions to the read-only state for a region. In TSP, this overhead is incurred within a critical section and has rippling effects on other processors waiting to enter that critical section, thereby causing a still larger difference in overall performance between the two region-based protocols (15%). The negligible difference in performance between the region trapping and annotated regions versions in all but one of these applications (TSP) suggests that region trapping costs play a minimal role in most cases. and that the significant difference in the performance between these region trapping applications and those based on VM primitives results from the differences in managing data at a region rather than page-granularity. These results demonstrate that the RTL can be used to eliminate the need for programmer annotations in such programs while maintaining efficient execution.

Compared to the VM version, the two region-based protocols improve performance in these applications by significant margins ranging from 18% in TSP to 41% in SOR on this platform. Using application-defined regions as the medium for sharing data reduces the number of traps that occur and the number of messages communicated in all six applications. Interestingly, however, the number of bytes communicated between processors is significantly higher in the region-based protocols in two of the applications, LU and TSP. While a better choice of regions might improve this situation, the increase in bytes using regions in these two applications is a result of defining large regions that span multiple virtual memory pages. These applications suffer from false sharing within regions, where modifications to the entire region are transmitted between processors on a trap even though much of region may not be used by the other processor. On the SGI platform, the reduction in the number of messages communicated between processors in these two applications compensates for the increase in the number of bytes transmitted.

It is worth noting that in our environment the RT and AN versions of MM execute significantly faster than the VM version. This is rather surprising, since other studies report near-linear speedup for page-based DSM implementations of MM. We do not obtain near-linear speedup for the VM version of MM because we use a matrix size that results in false sharing, our execution times include the time required to fault all data to remote machines, and IRIX 6.2 appears to delay the delivery of SIGIO signals to an executing process until it either blocks or its quantum expires. In some cases, this results in delays when requesting remote pages or regions. While such delays are not present in other environments we've used in previous studies we have found that the AN version of MM still outperforms the VM version (although in this case, the execution time is only improved by 12%) [20]. This earlier publication [20] also provides a more detailed discussion comparing the performance of page-based (VM) and region-based protocols (AN).

5.3 Case Study Summary

These results provide some evidence that the overhead of the region trapping mechanisms within a DSM environment are reasonable. Since trapping overheads account for a small proportion of the execution time for both RT and VM mechanisms, the key factor in determining whether region trapping is useful within the distributed shared memory context lies in the trade-offs between using regions rather than pages for managing shared data, both from a programming and performance perspective. Studies by Adve *et al.* [1] and by Buck and Keleher [7] compare the performance of page-based LRC to object-based EC. Each study suggests that page-based systems are competitive with and sometimes better than object-based systems, while a similar study by Neves *et al.* [19] has found object-based systems to be much better. However, unlike the page and region-based versions of LRC used in the study presented in this section, page-based LRC and object-based EC differ not only in the use of pages rather than objects (regions) as the granularity of data management, but also in terms of the synchronization model (release consistency versus entry consistency) and coherence protocols (lazy release consistency versus write-update) that are used. Consequently, the results of those studies are not directly comparable to those presented in this section.

The comparison between the page and region-based versions of LRC presented in this section suggest that many applications may benefit from using these region trapping mechanisms instead of the traditional page-based VM mechanisms. However, a detailed examination of these trade-offs between pages and regions within a distributed shared memory system are beyond the scope of the study presented in this section. Consequently, a number of factors that may also influence the choice of whether to use pages or regions for managing shared data have not been considered here. This includes the use of page aggregation techniques, which increase the effective size of a page [3] and would likely improve the performance of the VM case for some of the coarse-grained applications used in this study, and the use of other coherence protocols such as scope consistency [12], which captures some of the advantages of object-based protocols such as entry consistency.

6 Discussion

The benchmarks and case study of Sections 4 and 5 highlight both the potential and the limitations of the current RTL implementation. The primary limitations, both in terms of the programming interface and performance, can, as it turns out, be easily addressed. In this section, we briefly discuss these limitations and how they can be overcome in the RTL.

One of the key performance costs in the RTL, the need to update the read-only copy of the RTL, can be eliminated by mapping the read-only and read-write memory areas to the *same* physical memory area. Once this is done, all of the *RT-update* costs described in Section 3.3 and shown in the figures of Section 4 are eliminated since only a single physical copy of the region needs to be maintained (unless a copy is required in order to compute *diffs*). This also eliminates the additional physical memory overhead incurred by the current RTL implementation.

The other major concern is with the RTL interface itself, which requires all pointers to a region to be explicitly declared. In work conducted concurrently and independently of our own, Itzkovitz and Schuster [11] present an alternative approach that provides the functionality that the RTL implements but without having to manipulate pointers to those regions. Itzkovitz and Schuster's MultiView system allocates each region on a separate virtual memory page, but maps each of those regions to the same set of physical memory pages. This allows the virtual memory protection levels of each region to be manipulated independently while still allocating different regions on the same page.

The MultiView approach presents a simpler programming paradigm than the RTL (regions must still be identified, but pointers to regions do not). However, MultiView may also consume a significant portion of the virtual address space because a single virtual memory page must be allocated for each region, even if the region is only a few bytes in size. This leads to a potentially more serious performance drawback in that each virtual memory page requires a single TLB entry. Since most current architectures have relatively limited TLB sizes, applications that reference many small regions may generate significantly more TLB misses using the MultiView approach. In contrast the RTL approach has a small fixed virtual memory and TLB entry overhead that is dependent only on the total size of the virtual memory consumed by the application. Further, once the RT-update costs in the RTL have been eliminated using the technique noted above, changing RTL protection levels will be a fraction of the cost of manipulating virtual memory protection levels. These latter costs, using the expensive `mprotect` system call, are required both in traditional page-based approaches and by MultiView. Thus while the MultiView approach offers clear advantages as far as the programming interface is concerned these performance issues may make the RTL approach more suitable for very fine-grained sharing.

7 Conclusions

In this paper, we have described the design and implementation of the Region Trap Library, which provides the same functionality, at the granularity of user-defined regions, that application domains typically require from virtual memory primitives at a page-granularity. One of the main contributions of the mechanisms used in this library is that they do not depend on features of a particular programming language and they are portable across several architectures. Benchmarks on several operating systems and architectures suggests that the overhead of these mechanisms is typically competitive with their page-based counterparts. Our implementation of a region-based version of the *lazy release consistency* coherence protocol within the TreadMarks page-based DSM system demonstrates the applicability of region trapping mechanisms within some of the domains that make use of virtual memory primitives. In the DSM context, we found region trapping overheads to be typically less than 1%, with the exception of one application that incurred an overhead of 6%.

Together, the benchmark results and the DSM case study suggest that the region trapping mechanisms implemented in the Region Trap Library provide a feasible region-granularity data management alternative to VM primitives within some of the application domains that commonly rely on page-based VM primitives. Further study is needed to identify the overhead of the region trapping mechanisms within some of these other application domains, and to determine the value of using regions rather than pages as the unit of data management in these domains.

Acknowledgments

The authors gratefully acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC). We thank Graham Smith for porting the RTL to LINUX on the Pentium Pro and Diego Moscoso for discussions during the early stages of this work. We thank Ken Sevcik and the POW group at the University of Toronto for providing us with access to their IBM systems. Finally we wish to thank the anonymous referees for their careful reading of the paper and for their insightful comments.

References

[1] S.V. Adve, A.L. Cox, S. Dwarkadas, R. Rajamony and W. Zwaenepoel, "A Comparison of Entry Consistency and Lazy Release Consistency Implementations", Proceedings of the 2nd International Symposium on High-Performance Computer Architecture, February, 1996.

[2] A. Agarwal, B-H. Lim, D. Kranz, and J. Kubiatowicz, "APRIL: A processor architecture for multiprocessing", Proceedings of the 17th International Symposium on Computer Architecture, pp. 104-114, May, 1990.

[3] C. Amza, A.L. Cox, K. Rajamani, and W. Zwaenepoel, "Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory", Proceedings of the Sixth Conference on Principles and Practice of Parallel Programming, pp. 90-99, June 1997.

[4] A.W. Appel and K. Li, "Virtual Memory Primitives for User Programs", Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 96-107, April, 1991.

[5] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, "Orca: A language for parallel programming of distributed systems", IEEE Transactions on Software Engineering pp. 190-205, March, 1992.

[6] B. Bershad, M. Zekauskas and W. Sawdon, "The Midway Distributed Shared Memory System", Proceedings of COMPCOM '93, pp. 528-537, February, 1993.

[7] B. Buck and P. Keleher, "Locality and Performance of Page- and Object-Based DSMs", Proceedings of the 12th International Parallel Processing Symposium, March, 1998.

[8] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Lettlefield, "The Amber System: Parallel programming on a network of multiprocessors", Proceedings of the 12th ACM Symposium on Operating Systems Principles, pp. 147-158, December, 1989.

[9] M. Feeley and H. Levy, "Distributed Shared Memory with Versioned Objects", Conference

on Object-Oriented Programming Systems Languages, and Applications, October, 1992.

[10] A.L. Hosking and J.E.B. Moss, "Protection Traps and Alternatives for Memory Management of an Object-Oriented Language", Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, pp. 106-119, December 1993.

[11] A. Itzkovitz and A. Schuster, "MultiView and Millipage – Fine-Grain Sharing in Page-Based DSMs", Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99), February, 1999.

[12] L. Iftode, J.P. Singh, and K. Li, "Scope Consistency: A Bridge between Release Consistency and Entry Consistency", Proceedings of the Symposium on Parallel Algorithms and Architectures, June 1996.

[13] K. Johnson, F. Kaashoek and D. Wallach, "CRL: High-Performance All Software Distributed Shared Memory", Proceedings of the 15th Symposium on Operating Systems Principles, pp. 213-228, December, 1995.

[14] P. Keleher, A. Cox, S. Dwarkadas and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", Proceedings of Winter 1995 USENIX Conference, pp. 115-131, 1994.

[15] C. Lamb, G. Landis, J. Orenstein and D. Weinreb, "The Objectstore Database System", Communications of the ACM, Vol. 34, No. 10, pp. 50-63, October, 1991.

[16] C. Lakos, "Implementing BCPL on the Burroughs B6700", Software Practices and Experience, Vol. 10, pp. 673-683, 1980.

[17] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems", ACM Transactions on Computer Systems, Vol. 7, No. 4, pp. 321-359, November, 1989.

[18] J.E.B. Moss "Working with Persistent Objects: To Swizzle or not to Swizzle", IEEE Transactions on Software Engineering, Vol. 18, No. 8 pp. 657-673, August, 1992.

[19] N. Neves, M. Castro, and P. Guedes, "A Checkpoint Protocol for an Entry Consistent Shared Memory System", Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing August, 1994.

[20] H. Sandhu, T. Brecht, and D. Moscoso, "Multiple-Writer Entry Consistency", International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98), pp. 355-362, July, 1998.

[21] D. J. Scales and M.S. Lam, "The Design and Evaluation of a Shared Object System for Distributed Memory Machines", Proceedings of the First Symposium on Operating System Design and Implementation, pp. 101-114, November, 1994.

[22] D.J. Scales, K. Gharachorloo, and C.A. Thekkath, "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory", Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems, October, 1996.

[23] I. Schoinas, B. Falsafi, A.R. Lebek, S.K. Reinhardt, J.R. Larus, and D.A. Wood, "Fine-Grain Access Control for Distributed Shared Memory", Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 297-306, October, 1994.

[24] V. Singhal, S.V.Kakkad, and P.R.Wilson, "Texas: An Efficient, Portable Persistent Store", Proceedings of the Fifth Int'l. Workshop on Persistent Object Systems, September 1992.

[25] C.A. Thekkath and H. M. Levy, "Hardware and Software Support for Efficient Trap Handling", Proceedings of ASPLOS-IV, October 1994.

[26] S. T. White, "Pointer Swizzling Techniques for Object-Oriented Database Systems", Ph.D. Thesis, University of Wisconsin, 1994.

[27] S. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations", Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 24-36, 1995.

[28] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad, "Software Write Detection for Distributed Shared Memory", Proceedings of the First Symposium on Operating System Design and Implementation, pp. 87-100, November, 1994.

The Case for Compressed Caching in Virtual Memory Systems

Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis

Dept. of Computer Sciences
University of Texas at Austin
Austin, Texas 78751-1182
{wilson|sfkaplan|smaragd}@cs.utexas.edu
http://www.cs.utexas.edu/users/oops/

Abstract

Compressed caching uses part of the available RAM to hold pages in compressed form, effectively adding a new level to the virtual memory hierarchy. This level attempts to bridge the huge performance gap between normal (uncompressed) RAM and disk.

Unfortunately, previous studies did not show a consistent benefit from the use of compressed virtual memory. In this study, we show that technology trends favor compressed virtual memory—it is attractive now, offering reduction of paging costs of several tens of percent, and it will be increasingly attractive as CPU speeds increase faster than disk speeds.

Two of the elements of our approach are innovative. First, we introduce novel compression algorithms suited to compressing in-memory data representations. These algorithms are competitive with more mature Ziv-Lempel compressors, and complement them. Second, we adaptively determine how much memory (if at all) should be compressed by keeping track of recent program behavior. This solves the problem of different programs, or phases within the same program, performing best for different amounts of compressed memory.

1 Introduction

For decades, CPU speeds have continued to double every 18 months to two years, but disk latencies have improved only very slowly. Disk latencies are five to six orders of magnitude greater than main memory access latencies, while other adjacent levels in the memory hierarchy typically differ by less than one order of magnitude. Programs that run entirely in RAM benefit from improvements in CPU speeds, but the runtime of programs that page is likely to be dominated by disk seeks, and may run many times more slowly than CPU-bound programs.

In [Wil90, Wil91b] we proposed compressed caching for virtual memory—storing pages in compressed form in a main memory *compression cache* to reduce disk paging. Appel also promoted this idea [AL91], and it was evaluated empirically by Douglis [Dou93] and by Russinovich and Cogswell [RC96]. Unfortunately Douglis's experiments with Sprite showed speedups for some programs, but no speedup or some slowdown for others. Russinovich and Cogswell's data for a mixed PC workload showed only a slight potential benefit. There is a widespread belief that compressed virtual memory is attractive only for machines without a fast local disk, such as diskless handheld computers or network computers, and laptops with slow disks. As we and Douglis pointed out, however, compressed virtual memory is more attractive as CPUs continue to get faster. This crucial point seems to have been generally overlooked, and no operating system designers have adopted compressed caching.

In this paper, we make a case for the value of compressed caching in modern systems. We aim to show that the discouraging results of former studies were primarily due to the use of machines that were quite slow by current standards. For current, fast, disk-based machines, compressed virtual memory offers substantial performance improvements, and its advantages only increase as processors get faster. We also study future trends in memory and disk bandwidths. As we show, compressed caching will be increasingly attractive, regardless of other OS improvements (like sophisticated prefetching policies, which reduce the average cost of disk seeks, and log-structured file systems, which reduce the cost of writes to disk).

We will also show that the use of better compression algorithms can provide a significant further improvement in the performance of compressed caching. Better Ziv-Lempel variants are now available, and we introduce here a new family of compression algorithms designed for in-memory data representations rather than file data.

The concrete points in our analysis come from simulations of programs covering a variety of memory requirements and locality characteristics. At this stage of our experiments, simulation was our chosen method of evaluation because it allowed us to easily try many ideas in a controlled environment. It should be noted that all our simulation parameters are either relatively conservative or perfectly realistic. For instance, we assume a quite fast disk in our experiments. At the same time, the costs of compressions and decompressions used in our simulations are the actual runtime costs for the exact pages whose compression or decompression is being simulated at any time.

The main value of our simulation results, however, is not in estimating the exact benefit of compressed caching (even though it is clearly substantial). Instead, we demonstrate that it is possible to detect reliably how much memory should be compressed during a phase of program execution. The result is a compressed virtual memory policy that adapts to program behavior. The exact amount of compressed memory crucially affects program performance: compressing too much memory when it is not needed can be detrimental, as is compressing too little memory when slightly more would prevent many memory faults. Unlike any fixed fraction of compressed memory, our adaptive compressed caching scheme yields uniformly high benefits for all test programs and a wide range of memory sizes.

2 Compression Algorithms

In [WLM91] we explained how a compressor with a knowledge of a programming language implementation could exploit that knowledge to achieve high compression ratios for data used by programs. In particular, we explained how pointer data contain very little information on average, and that pointers can often be compressed down to a single bit.

Here we describe algorithms that make much weaker assumptions, primarily exploiting data regularities imposed by hardware architectures and common programming and language-implementation strategies. These algorithms are fast and fairly *symmetrical*—compression is not much slower than decompression. This makes them especially suitable for compressed virtual memory applications, where pages are compressed about as often as they're decompressed. [1]

[1]A variant of one of our algorithms has been used successfully for several years in the virtual memory system of the Apple Newton, a personal digital assistant with no disk [SW91] (Walter Smith, personal communication 1994, 1997). While we have not previously published this algorithm, we sketched it for Smith and he used it in the Newton, with good results—it achieved slightly less compression than a Ziv-Lempel algorithm Apple had used previously, but was much faster. Unfortunately, we do not have any detailed performance comparisons.

As we explain below, the results for these algorithms are quite encouraging. A straightforward implementation in C is competitive with the best assembly-coded Ziv-Lempel compressor we could find, and superior to the LZRW1 algorithm (written in C by Ross Williams)[Wil91a] used in previous studies of compressed virtual memory and compressed file caching.

As we will explain, we believe that our results are significant not only because our algorithms are competitive and often superior to advanced Ziv-Lempel algorithms, but because they are *different*. Despite their immaturity, they work well, and they complement other techniques. They also suggest areas for research into significantly more effective algorithms for in-memory data.

(Our algorithms are also interesting in that they could be implemented in a very small amount of hardware, including only a tiny amount of space for dictionaries, providing extraordinarily fast and cheap compression with a small amount of hardware support.)

2.1 Background: Compression

To understand our algorithms and their relationship to other algorithms, it is necessary to understand a few basic ideas about data compression. (We will focus on lossless compression, which allows exact reconstruction of the original data, because lossy compression would generally be a disaster for compressed VM.)

All data compression algorithms are in a deep sense *ad hoc*—they must exploit *expected regularities* in data to achieve any compression at all. All compression algorithms embody expectations about the kinds of regularities that will be encountered in the data being compressed. Depending on the kind of data being compressed, the expectations may be appropriate or inappropriate and compression may work better or worse. The main key to good compression is having the right kinds of expectations for the data at hand.

Compression can be thought of as consisting of two phases, which are typically interleaved in practice: *modeling* and *encoding* [BCW90, Nel95]. Modeling is the process of detecting regularities that allow a more concise representation of the information. Encoding is the construction of that more concise representation.

Ziv-Lempel compression. Most compression algorithms, including the overwhelmingly popular Ziv-Lempel family, are based on detection of *exact* repetitions of *strings* of atomic tokens. The token size is usually one byte, for speed reasons and because much data is in some sense byte-oriented (e.g., characters in a text file) or multiple-byte oriented (e.g., some kinds of image data, Intel Architecture machine code, unicode).

A Ziv-Lempel compressor models by reading through the input data token by token, constructing a dictionary of observed sequences, and looking for repetitions as it goes. It encodes by writing strings to its output the first time they are observed, but writing special codes when a repetition is encountered (e.g., the number of the dictionary entry). The output thus consists of appropriately labeled "new" data and references to "old" data (repetitions).

The corresponding LZ decompressor reads through this data much like an interpreter, reconstructing the dictionary created during compression. When it sees a new string, it adds it to the dictionary just as the compressor did, as well as sending it to its (uncompressed) output. When it sees a code for a repetition of a dictionary item, it copies that item to its output. In this way, its dictionary always matches the dictionary that the compressor had at the same point in the data stream, and its output replicates the original input by expanding the repetition codes into the strings they represent.

The main assumption embodied by this kind of compressor is that literal repetitions of multi-token strings will occur in the input—e.g., you'll often see several bytes in a row that are exactly the same bytes in the same order as something you saw before. This is a natural assumption in text, and reasonable in some other kinds of data, but often wrong for in-memory data.

2.2 In-Memory Data Representations

It is commonly thought that LZ-style compression is "general purpose," and that in-memory data are fairly arbitrary—different programs operate on different kinds of data in different ways, so there's not much hope for a better algorithm than LZ for compressing in-memory data. The first assumption is basically false,[2] and the second is hasty, so the conclusion is dubious.

While different programs do different things, there are some common regularities, which is all a compression algorithm needs to work well on average. Rather than consisting of byte strings, the data in memory are often best viewed as records and data structures—the overall array of memory words is typically used to store records, whose fields are mostly one or two words. Note that fields of records are usually *word-aligned* and that the data in those words are frequently numbers or pointers. Pointers can be usefully viewed *as* numbers—they are integer indices into the array of memory itself.

[2]It is worth stressing this again, because there is widespread confusion about the "optimality" of some compression algorithms. In general, an *encoding* scheme (such as Huffman coding or arithmetic coding) can be provably optimal within some small factor, but a compressor cannot, unless the regularities in the data are known in advance and in detail. Sometimes compression algorithms are proven optimal based on the simplifying assumption that the source is a stochastic (randomized, typically Markov) source, but real data sources in programs are generally *not* stochastic[WJNB95], so the proof does not hold for real data.

Integer and pointer data often have certain strong regularities. Integer values are usually numerically small (so that only their low-order bytes have significant information content), or else similar to other integers very nearby in memory.

Likewise, pointers are likely to point to other objects nearby in memory, or be similar to other nearby pointers—that is, they may point to another area of memory, but other pointers nearby may point to the same area. These regularities are quite common and strong. One reason is that heap data are often well-clustered; common memory allocators tend to allocate mostly within a small area of memory most of the time; data structures constructed during a particular phase of program execution are often well-clustered and consist of one or a few types of similar objects [WJNB95].

Other kinds of data often show similar regularities. Examples include the hidden headers many allocators put on heap objects, virtual function table pointers in C++ objects, booleans, etc.

These regularities are strong largely because in-memory data representations are designed primarily for speed, not space, and because real programs do not usually use random data or do random things with data. (Even randomized data can be very regular in this way; consider an array of random integers less than 1000—all of them will have zeroes in their upper 22 bits.)

2.3 Exploiting In-Memory Data Regularities

Our goal in this section is to convey the basic flavor of our algorithms (which we call WK algorithms); the actual code is available from our web site and is well-commented for those who wish to explore it or experiment with it.

We note that these algorithms were designed several years ago, when CPU's were much slower than today—they therefore stress simplicity and speed over achieving high compression. We believe that better algorithms can be designed by refining the basic modeling technique, perhaps in combination with more traditional sequence-oriented modeling, and by using more sophisticated encoding strategies. Given their simplicity, however, they are strikingly effective in our experiments.

Our compression algorithms exploit in-memory data regularities by scanning through the input data a 32-bit *word* at a time, and looking for data that are *numerically* similar—specifically, repetitions of the *high-order* 22-bit pattern of a word, even if the low-order 10 bits are different.[3] They therefore perform *partial* matching of whole-word bit patterns.

To detect repetitions, the encoder maintains a dictionary of just 16 *recently-seen words*. (One of our algorithms manages this dictionary as a direct mapped cache, and another as a 4x4 set-associative cache, with LRU used as the replacement algorithm for each set. These are simple software caching schemes, and could be trivially implemented in very fast hardware. Due to lack of space and because the exact algorithm did not matter for compressed caching performance, we will only discuss the direct-mapped algorithm in this paper.)

For these compression algorithms to work as well as they do, the regularities must be very strong. Where a typical LZ-style compressor uses a dictionary of many kilobytes (e.g., 64 KB), our compressors use only 64 *bytes* and achieve similar compression ratios for in-memory data.

The compressor scans through a page, reading each word, probing its cache (dictionary) for a matching pattern, and emitting a two-bit code classifying the word. A word may

- not match a dictionary entry, or
- match only in the upper 22 bits, or
- match a whole 32-bit pattern.

As a special case, we check first to see if the word is all zeroes, i.e., matches a full-word zero, in which case we use the fourth two-bit pattern.

For the all-zeroes case, only the two-bit tag is written to the compressed output page. For the other three cases, additional information must be emitted as well. In the no-match case, the entire 32-bit pattern that did not match anything is written to the output. For a full (32-bit) match, the dictionary index is written, indicating which dictionary word was repeated. For the partial (22-bit) match case, the dictionary index and the (differing) low 10 bits are written.

The corresponding decompressor reads through the compressed output, examining one two-bit tag at a time and taking the appropriate action. As with more conventional compression schemes, a tag indicating no-match directs it to read an item (one word) from the compressed input, insert it in the dictionary, and echo it to the output. A tag indicating all-zeroes directs it to write a word of zeroes to its output. A tag indicating a full-word match directs it to copy a dictionary item to the output, either whole (in the full match case) or with its low bits replaced by bits consumed from the input (for a partial match).

The encoding can then be performed quickly. Rather than actually writing the result of compressing a word directly to the output, the algorithm writes each kind of information into a different intermediate array as it reads through the input data, and then a separate postprocessing pass "packs" that information into the output page, using a fast packing routine. (The output page is segmented, with each segment containing one kind of data: tags, dictionary indices, low bits, and full words.) For example, the two-bit tags are actually written as bytes into a byte array, and a special routine packs four consecutive words (holding 16 tags) into a single word of output by shifting and XORing them together. During decompression, a prepass unpacks these segments before the main pass reconstructs the original data.

3 Adaptively Adjusting the Compression Cache Size

To perform well, a compressed caching system should adapt to the working set sizes of the programs it caches for. If a program's working set fits comfortably in RAM, few pages (or no pages) should be kept compressed, so that the overwhelming majority of pages can be kept in uncompressed form and accessed with no penalty. If a program's working set is larger than the available RAM, and compressing pages would allow it to be kept in RAM, more pages should be compressed until the working set is "captured". In this case, the reduction in disk faults may greatly outweigh the increase in compression cache accesses, because disk faults are many times more expensive than compression cache faults.

Douglis observed in his experiments that different programs needed compressed caches of different sizes. He implemented an adaptive cache-sizing scheme, which varied the split between uncompressed and compressed RAM dynamically. Even with this adaptive caching system, however, his results were inconsistent; some programs ran faster, but others ran slower. We believe that Douglis's adaptive caching strategy may have been partly at fault. Douglis used a fairly simple scheme in which the two caches competed for RAM on the basis of how

[3] The 22/10 split was arrived at experimentally, using an early data set that partially overlaps the one used in this study. The effectiveness of the algorithm is not very sensitive to this parameter, however, and varying the split by 2 bits does not seem to make much difference—using more high bits means that matches are encoded more compactly, but somewhat fewer things match.

recently their pages were accessed, rather like a normal global replacement policy arbitrating between the needs of multiple processes, keeping the most recently-touched pages in RAM. Given that the uncompressed cache *always* holds more recently-touched pages than the compressed cache, this scheme requires a bias to ensure that the compressed cache has any memory at all. We believe that this biased recency-based caching can be maladaptive, and that a robust adaptive cache-sizing policy *cannot* be based solely on the LRU ordering of pages *within* the caches.

3.1 Online Cost/Benefit Analysis

Our own adaptive cache-sizing mechanism addresses the issue of adaptation by performing an online cost/benefit analysis, based on recent program behavior statistics. Assuming that behavior in the relatively near future will resemble behavior in the relatively recent past, our mechanism actually keeps track of aspects of program behavior that bear directly on the performance of compressed caching for different cache sizes, and compresses more or fewer pages to improve performance.

This system uses the kind of recency information kept by normal replacement policies, i.e., it maintains an approximate ordering of the pages by how recently they have been touched. Our system extends this by retaining the same information for pages which have been recently evicted. This information is discarded by most replacement policies, but can be retained and used to tell *how well* a replacement policy is working, compared to what a *different* replacement policy would do.

We therefore maintain an LRU (or *recency*) ordering of the pages in memory *and* a comparable number of recently-evicted pages. This ordering is not used primarily to model what *is* in the cache, but rather to model *what the program is doing*.

A Simplified Example. To understand how our system works, consider a very simple version which manages a pool of 100 page frames, and only chooses between two compressed cache sizes: 50 frames, and 0 frames. With a compression cache of 50 frames and a compression ratio of 2:1, we can hold the 50 most-recently-accessed pages in uncompressed form in the uncompressed cache, and the next 100 in compressed form. This effectively increases the size of our memory by 50% in terms of its effect on the disk fault rate.

The task of our adaptation mechanism is to decide whether doing this is preferable to keeping 100 pages in uncompressed form and zero in compressed form. (We generally assume that pages are compressed before being evicted to disk, whether or not the compression cache is of significant size. Our experiments show that this cost is very small.)

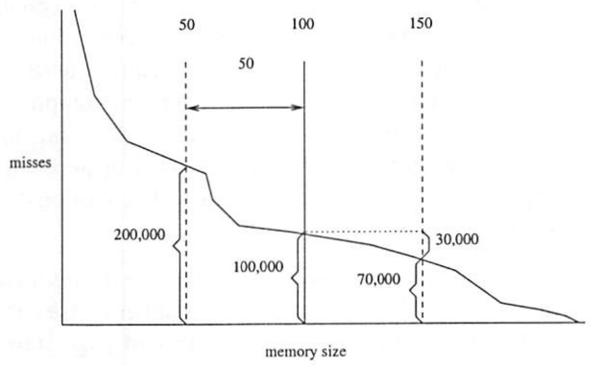


Figure 1: Cost/benefit computation using the miss-rate histogram.

Figure 1 shows an example miss rate histogram decorated with some significant data points. (This is not real data, and not to scale because the actual curve is typically *very* high on the far left, but the data points chosen are reasonable).

The benefit of this 50/50 configuration is the reduction in disk faults in going from a memory of size 100 to a memory of size 150. We can measure this benefit simply by counting the number of times we fault on pages that are between the 101st and 150th positions in the LRU ordering (30,000 in Figure 1), and multiplying that count by the cost of disk service.

The cost of this 50/50 configuration is the cost of compressing and decompressing all pages outside the uncompressed cache region. These are exactly the touches to the pages beyond the 51st position in the LRU ordering (200,000 touches). Thus, in the example of Figure 1, compressed caching is beneficial if compressing and decompressing 200,000 pages is faster than fetching 30,000 pages from disk.

In general, our recency information allows us to estimate the cost and benefit of a compression cache of a given size, regardless of what the current size of the compression cache actually is, and which pages are currently in memory. That is, we can do a "what if analysis" to find out if the current split of memory between caches is a good one, and what might be a better one. We can simply count the number of touches to pages in different regions of the LRU ordering, and interpret those as hits or misses relative to different sizes of uncompressed cache and corresponding sizes of compressed cache and overall effective memory size.

Multiple target sizes. We generalize the above scheme by using several different "target" compression cache sizes, interpreting touches to different ranges of the LRU ordering appropriately for each size. The adaptive component of our system computes the costs and benefits of each of the target sizes, based on recent counts of touches to regions of the LRU ordering, and chooses the target size with the lowest cost. Then the compressed cache size is adjusted in a *demand-driven* way: memory is compressed or uncompressed only when an access to a compressed page (either in compressed RAM or on disk) occurs.

Actually, our system chooses a target *uncompressed* cache size, and the corresponding overall effective cache size is computed based on the number of page frames left for the compressed cache, multiplied by an estimate of the compression ratio for recently compressed pages. This means that the statistics kept by our adaptivity mechanism are not exact (our past information may contain hits that are in a different recency region than that indicated by the current compressibility estimate). Nevertheless, this does not seem to matter much in our simulations; approximate statistics about which pages have been touched how recently are quite sufficient. This indicates that our system will not be sensitive to the details of the replacement policy used for the uncompressed cache; any normal LRU approximation should work fine. (E.g., a clock algorithm using reference bits, a FIFO-LRU segmented queue using kernel page traps, or a RANDOM-LRU segmented queue using TLB miss handlers.)

The overheads of updating the statistics, performing the cost/benefit analyses, and adaptively choosing a target split are low—just a few hundred instructions per uncompressed cache miss, if the LRU list is implemented as a tree with an auxiliary table (a hash table or sparse page-table like structure).

3.2 Adapting to Recent Behavior

To adapt to recent program behavior our statistics are decayed exponentially with time. Time, however, is defined as the number of *interesting events* elapsed. Events that our system considers "interesting" are page touches that could affect our cost benefit analysis (i.e., would have been hits if we had compressed as much memory as any of our target compression sizes currently suggests). Defining time this way has the benefit that touches to very recently used pages are ignored, thus filtering out high-frequency events.

Additionally, the decay factor used is inversely proportional to the size of memory (total number of page frames), so that time typically advances more slowly for larger memories than for small ones—small memories usually have a shorter replacement cycle, and need to decay their statistics at a faster rate than larger ones.

If the decay rate were not inversely proportional to the memory size, time would advance inappropriately slowly for small memories and inappropriately quickly for large ones. The small cache would wait too long to respond to changes, and the large one would twitchily "jump" at brief changes in program behavior, which are likely not to persist long enough to be worth adapting toward.

Extensive simulation results show that this strategy works as intended: our adaptivity ensures that for any memory size, the cache responds to changes in the recent behavior of a program relatively quickly, so that it can benefit from relatively persistent program behavior, but not so quickly that it is continually "distracted" by short duration behaviors.

A single setting of the decay factor (relativized automatically to the memory size) works well across a variety of programs, and across a wide ranges of memory sizes.

4 Detailed Simulations

In this section, we describe the methodology and results of detailed simulations of compressed caching. We captured page image traces, recording the pages touched *and their contents*, for six varied UNIX programs, and used these to simulate compressed caching in detail.

(The code for our applications, tracing and filtering tools, and compressors and simulator are all available from our web site for detailed study and further research.)

Note that our traces do not contain references to executable code pages. We focus on data pages, because our main interest is in compressing in-memory data. As we will explain in Section 5, compressing code equally well is an extra complication but can certainly be done. Several techniques complementary to ours have been proposed for compressing code and the data from [RC96] indicate that references to code pages exhibit the same locality properties as references to data pages.

4.1 Methodology

Test suite. For these simulations, we traced six programs on an Intel x86 architecture under the Linux operating system with a page size of 4KB (we will study the effect of larger page sizes in Section 4.3). The behavior of most of these programs is described in more detail in [WJNB95]. Here is a brief description of each:

- **gnuplot**: A plotting program with a large input producing a scatter plot.

- **rscheme**: A bytecode-based interpreter for a garbage-collected language. Its performance is dominated by the runtime of a generational garbage collector.
- **espresso**: A circuit simulator.
- **gcc**: The component of the GNU C compiler that actually performs C compilation.
- **ghostscript**: A PostScript formatting engine.
- **p2c**: A Pascal to C translator.

These programs constitute a good test selection for locality experiments (as we try to test the adaptivity of our compressed caching policy relative to locality patterns at various memory sizes). Their data footprints vary widely: gnuplot and rscheme are large programs (with over 14,000 and 2,000 pages, respectively), gcc and ghostscript are medium-sized (around 550 pages), while espresso and p2c are small (around 100 pages).

We used the following three processors:

1. **Pentium Pro at 180 MHz**: This processor approximately represents an average desktop computer at this time. Compressed caching is not only for fast machines.
2. **UltraSPARC-10 300 Mhz**: While one of the fastest processors available now, it will be an average processor two years from now. Compressed caching works even better on a faster processor.
3. **UltraSPARC-2 168 MHz**: A slower SPARC machine which provides an interesting comparison to the Pentium Pro, due to its different architecture (e.g., faster memory subsystem).

We used three different compression algorithms in our experiments:

1. **WKdm**: A recency based compressor that operates on machine words and uses a direct-mapped, 16 word dictionary and a fast encoding implementation.
2. **LZO**: Specifically, **LZO1F**, is a carefully coded Lempel-Ziv implementation designed to be fast, particularly on decompression tasks. It is well suited to compressing small blocks of data, using small codes when the dictionary is small. While all compressors we study are written in C, this one also has a speed-optimized implementation (in Intel x86 assembly) for the Pentium Pro.
3. **LZRW1**: Another fast Lempel-Ziv implementation. This algorithm was used by Douglis in [Dou93]. While it does not perform as well as LZO, we wanted to demonstrate that even this algorithm would allow for an effective compressed cache on today's hardware.

The runtimes of the test suite. Our results are presented in terms of time spent paging, but it is helpful to know the processing time required to execute each program in the test suite. Figure 2 shows the time required to execute each of our six programs on each of the three processors, when no paging occurs. These times can be added with paging time information to obtain total turnaround time for a given architecture, memory size, and virtual memory configuration.

Program name	P-Pro 180MHz	SPARC 168MHz	SPARC 300MHz
gnuplot	46.89	32.99	20.61
rscheme	8.26	11.77	7.59
espresso	10.07	12.35	7.41
gcc	9.89	14.66	9.41
ghostscript	18.95	26.89	16.84
p2c	2.38	2.91	2.08

Figure 2: The processing times for each program in the test suite on each processor used in this study. If enough memory is available such that no paging occurs, these times will be the turnaround times.

A brief note on compressor performance. All of our compression algorithms achieve roughly a factor of two in compression on average for all six programs. All can compress and decompress a page in well under half a millisecond on all processors. The WKdm algorithm is the fastest, compressing a page in about 0.25 milliseconds and decompressing in about 0.15 milliseconds on the Pentium Pro, faster on the SPARC 168 MHz, and over twice as fast on the SPARC 300 MHz. (This is over 20 MB compressed *and* uncompressed per second, about the bandwidth of a quite fast disk.) LZO is about 20% slower, and LZRW1 about 20% slower still.

Tracing. Our simulator takes as input a trace of the pages a program touches, augmented with information about the compressibility and cost of compression of each touched page for a particular compression algorithm. To create such a trace and keep the trace size manageable, we used several steps and several tracing and filtering tools.

We traced each program using the portable tracing tool *VMTrace* [WKB]. We added a module to VMTrace that made it emit a complete copy of each page as it was referenced. We refer to such traces as *page image traces*.

Creating compression traces. To record the actual effectiveness and time cost of compressing each page image, we created a set of *compression traces*. For each combination of compression algorithm and CPU, we created a trace recording how expensive and how effective compression is for each page image in the reduced page image trace. Since we have 6 test programs, 3 compression algorithms, and 3 CPU's, this resulted in 54 compression traces.

The tool that creates compression traces is linked with a compressor and decompressor, and consumes a (reduced) page image trace. For each trace record in the page image trace, it compresses and decompresses the page image and outputs a trace record. This record contains the page number, the times for compressing and decompressing the page's contents at that moment, and the resulting compressed size of the page. Each page image is compressed and decompressed several times, and the median times are reported. Timing is very precise, using the Solaris high-resolution timer (all of our compression timings were done under the Solaris operating system). To avoid favorable (hardware) caching effects, the caches are filled with unrelated data before each compression or uncompression. (This is conservative, in that burstiness of page faults will usually mean that some of the relevant memory is still cached in the second-level cache in a real system.)

Simulation parameters. We used four different target compression sizes with values equal to 10%, 23%, 37%, and 50% of the simulated memory size. Thus, during persistent phases of program behavior (i.e., when the system has enough time to adapt) either none, or 10%, or 23%, or 37%, or 50% of our memory pages are holding compressed data. Limiting the number of target compression sizes to four guarantees that our cost/benefit analysis incurs a low overhead. The decay factor used is such that the M-th most recent event (with M being the size of memory) has a weight equal to 20% of the most recent event. Our results were not particularly sensitive to the exact value of the decay factor.

Estimates used. During simulation we had to estimate the costs for reading a page from disk or writing it to disk. We conservatively assumed that writing "dirty" pages to disk incurs no cost at all, to compensate for file systems that keep low the cost of multiple writes (e.g., log-structured file systems). Additionally, we assumed a disk with a uniform seek time of 5ms. Admittedly, a more complex model of disk access could yield more accurate results, but this should not affect the validity of our simulations (a 5ms seek time disk is fast by modern standards). In Section 4.3 we examine the effect of using a faster disk (up to a seek time of 0.625ms).

4.2 Results of Detailed Simulations

4.2.1 Wide Range Results

For each of our test programs, we chose a wide range of memory sizes to simulate. The plots of this section show the entire simulated range for each program. Subsequent sections, however, concentrate on the *interesting* region of memory sizes. This range usually begins around the size where a program spends 90% of its time paging and 10% of its time executing on the CPU, and ends at a size where the program causes very little paging.

Figure 3 shows log-scale plots of the paging time of each of our programs as a function of the memory size. Each line in the plot represents the results of simulating a compressed cache using a particular algorithm on our SPARC 168 MHz machine. The paging time of a regular LRU memory system (i.e., with no compression) is shown for a comparison. As can be seen, compressed caching yields benefits for a very wide range of memory sizes, indicating that our adaptivity mechanism reliably detects locality patterns of different sizes. Note that all compression algorithms exhibit benefits, even though there are definite differences in their performance.

Figure 3 only aims at conveying the general idea of the outcome of our experiments. The same results are analyzed in detail in subsequent sections (where we isolate interesting memory regions, algorithms, architectures, and trends).

4.2.2 Normalized Benefits and the Effect of Compression Algorithms

Our first goal is to quantify the benefits obtained by using compressed caching and to identify the effect of different compression algorithms on the overall system performance. It is hard to see this effect in Figure 3, which seems to indicate that all compression algorithms obtain similar results.

A more detailed plot reveals significant variations between algorithm performance. Figure 4 plots the normalized paging times for different algorithms in the interesting region. (Recall that this usually begins at the size where a program spends 90% of its time paging and 10% of its time executing on the CPU, and ends at a size where the program causes very little paging). By "normalized paging time" we mean the ratio of paging time for compressed caching over the paging time for a regular LRU

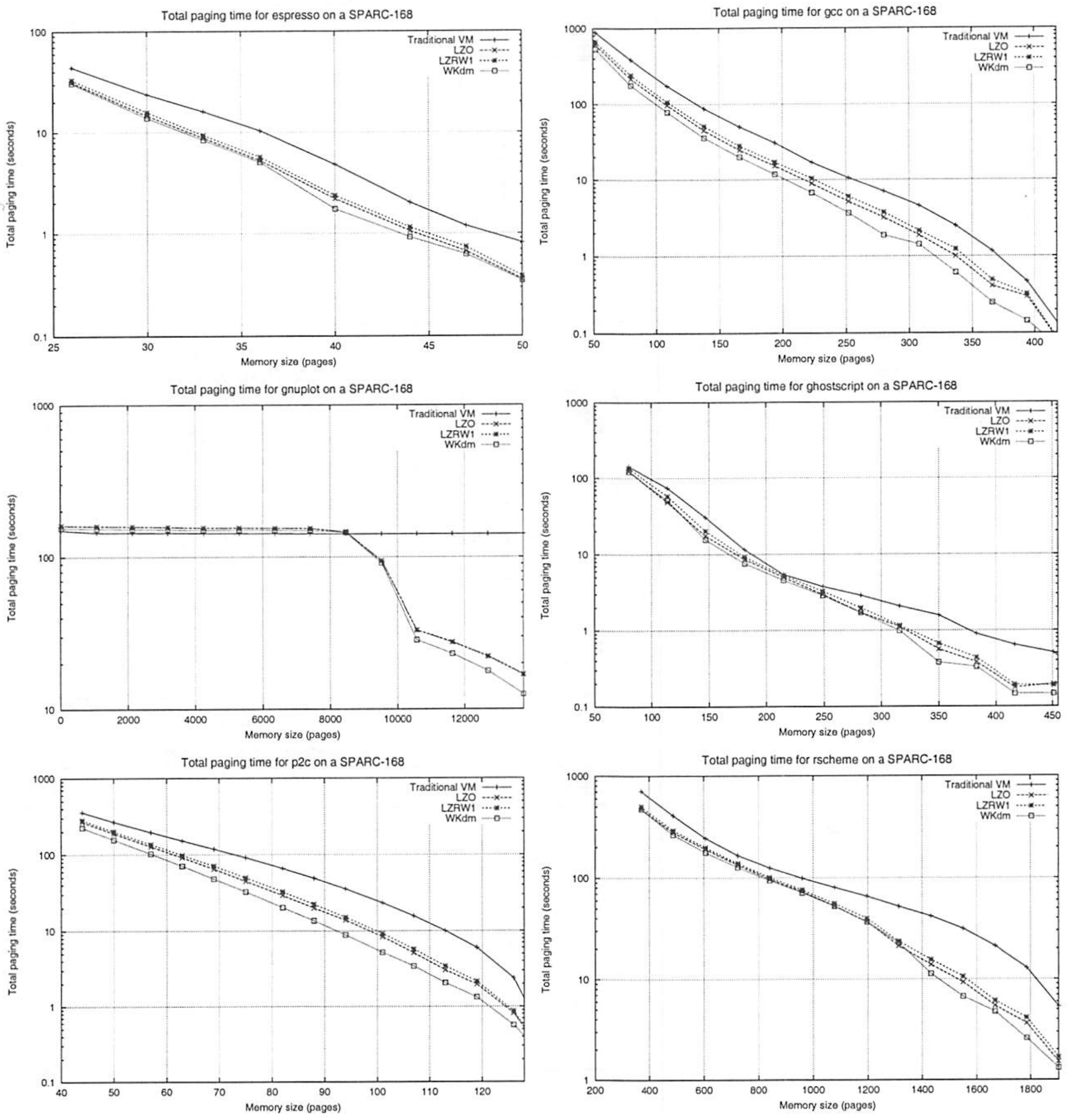


Figure 3: Compressed caching yields consistent benefits across a wide range of memory sizes.

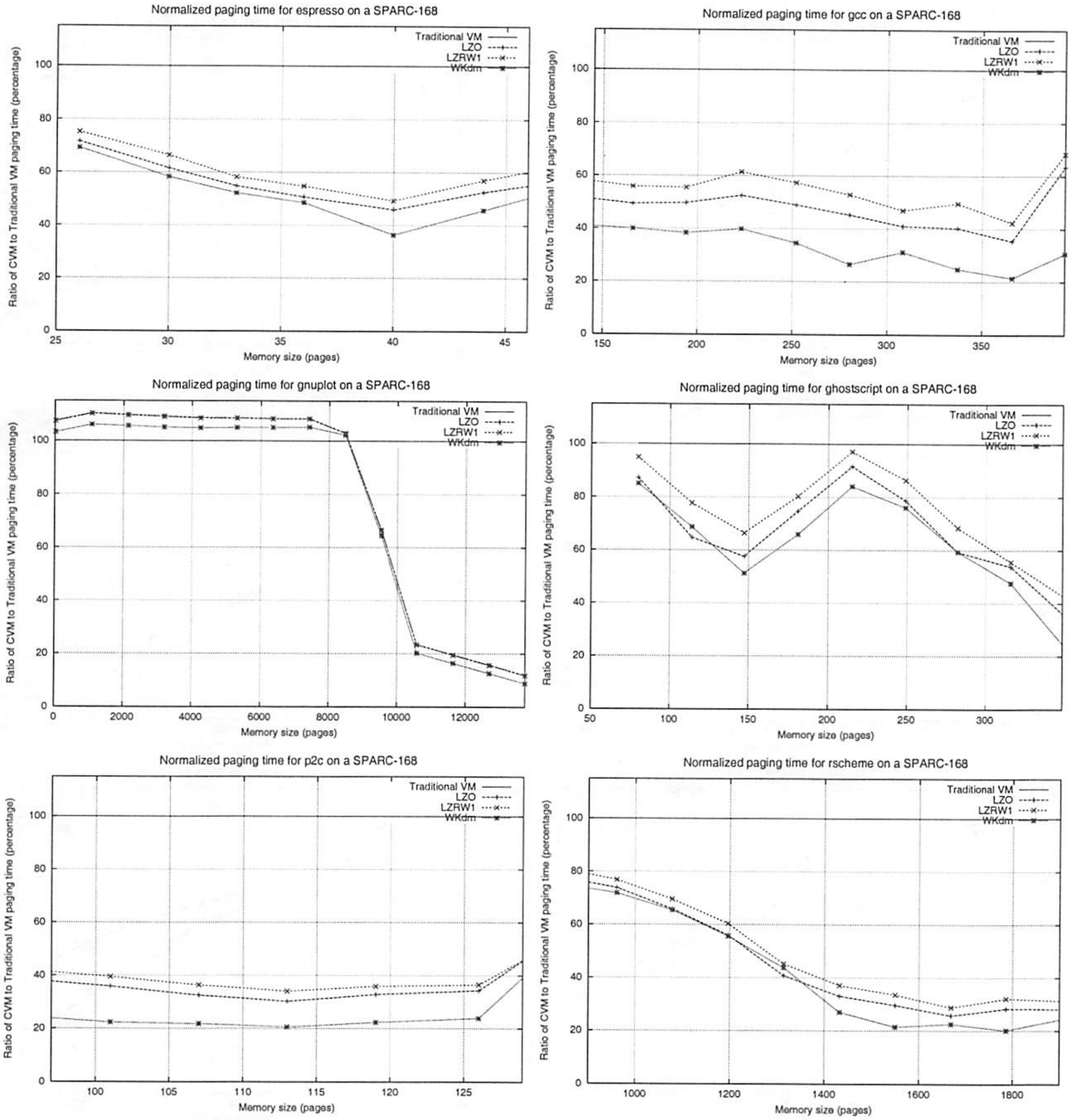


Figure 4: Varying compression algorithms can affect performance significantly. Even though all algorithms yield benefits compared to uncompressed virtual memory, some are significantly better than others.

replacement policy.

As can be seen, all algorithms obtain significant benefit over uncompressed virtual memory for the interesting ranges of memory sizes. Benefits of over 40% are common for large parts of the plots in Figure 4. At the same time, losses are rare (only exhibited for gnuplot) and small. Additionally, losses diminish for faster compression algorithms (and faster processors, which is not shown in this plot). That is, when our adaptivity does not perform optimally, its cost can be reduced by having a fast compression algorithm, since it is a direct function of performing unnecessary compressions and decompressions.

Gnuplot is an interesting program to study more closely. The program stores data that are highly compressible (exhibiting a ratio of over 4:1 on average). This way, the compressed VM policy can look at quite large memory sizes, expecting that it can compress enough pages so that all the required data remains in memory. Nevertheless, gnuplot's running time is dominated by a large loop iterating only twice on a lot of data. Hence, for small memory sizes the behavior that the compressed caching policy tries to exploit ends before any benefits can be seen. For larger sizes, the benefit can be substantial, reaching over 80%.

As shown in Figure 4, the performance difference of compressed caching under different compression algorithms can often be over 15%. Our WKdm algorithm achieves the best performance for the vast majority of data points, due to its speed and comparable compression rates to LZO. The LZRW1 algorithm, used by Douglis yields consistently the worst results. This fact, combined with the slow machine used (for current standards) are at least partially responsible for the rather disappointing results that Douglis observed.

4.2.3 Implementation and Architecture Effects

In the past sections we only showed results for our SPARC 168 MHz machine. As expected, the faster SPARC 300 MHz machine has a lower compression and decompression overhead and, thus, should perform better overall. The Pentium Pro 180 MHz machine is usually slower than both SPARC machines in compressing and uncompressing pages (not unexpectedly as it is an older architecture—see also out later remarks on memory bandwidth).

Figure 5 shows three of our test programs simulated under WKdm and LZO in all three architectures. For WKdm, the performance displayed agrees with our observations on machine speeds. Nevertheless, the performance of LZO is significantly better on the Pentium Pro 180 MHz machine than one would expect based on the machine speed alone. The reason is that, as pointed out earlier, the implementation of LZO we used on the Pentium Pro is hand optimized for speed in Intel x86 assembly language. Perhaps surprisingly, the effect of the optimization is quite significant, as can be seen. For ghostscript, for instance, the Pentium Pro is faster than the SPARC 168 MHz using LZO.

4.3 Technology Trends

4.3.1 Is Memory Bandwidth a Problem?

Compressed caching mostly benefits from the increases of CPU speed relative to disk latency. Nevertheless, a different factor comes into play when disk and memory *bandwidths* are taken into account. A first observation is that moving data from memory takes at most one-third of the execution time of our WKdm compression algorithm. (This ratio is true for both the Pentium Pro 180 MHz machine, which has a slow memory subsystem, and the SPARC 300 MHz, which has a fast processor. It is significantly better for the SPARC 168 MHz machine.) Hence, memory bandwidth does not seem to be the limiting factor for the near future. Even more importantly, faster memory architectures (e.g., RAMBUS) will soon become widespread and compression algorithms can fully benefit as they only need to read contiguous data. The overall trend is also favorable. Memory bandwidths have historically grown at 40%, while disk bandwidths and latencies have only grown at rates around 20%. (An analysis of technology trends can be found in M. Dahlin's "Technology Trends" Web Page at http://www.cs.utexas.edu/users/dahlin/techTrends/ .)

4.3.2 Sensitivity Analysis

The cost and benefits of compressed caching are dependent on the relative costs of compressing (and uncompressing) a page vs. fetching a page from disk. If compression is insufficiently fast relative to disk paging, compressed virtual memory will not be worthwhile.

On the other hand, if CPU speeds continue to increase far faster than disk speeds, as they have for many years, then compressed virtual memory will become increasingly effective and increasingly attractive. Over the last decade, CPU speeds have increased by about 60% a year, while disk latency and bandwidth have increased by only about 20% a year. This works out to an increase in CPU speeds *relative to disk speeds* of one third a year—or a doubling every two and a half years, and a quadrupling every five years.

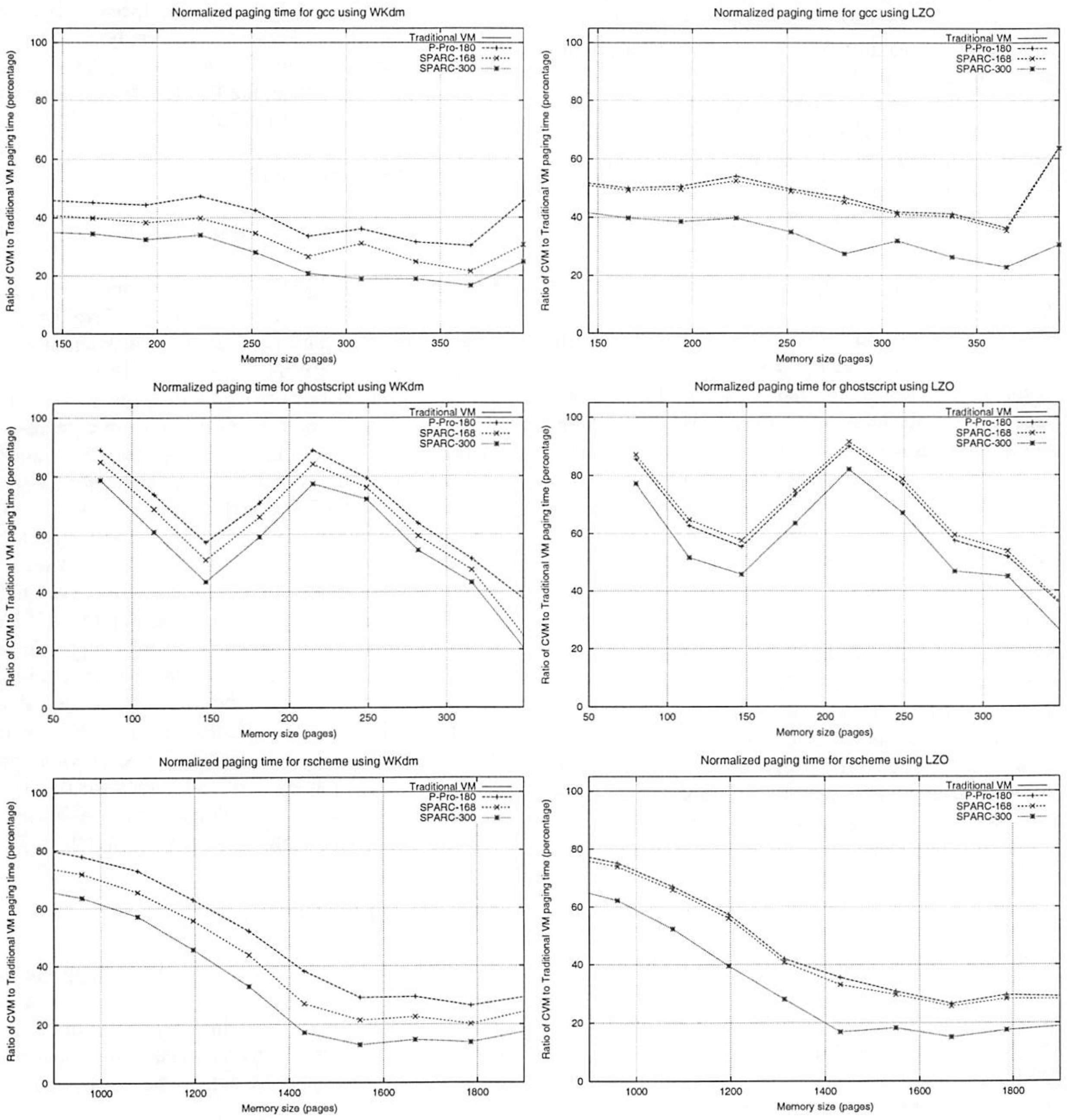


Figure 5: A SPARC 168 MHz usually has better performance than a Pentium Pro 180 MHz, while a SPARC 300 MHz is significantly better than both. Nevertheless, the Pentium Pro 180 MHz is much faster for a hand-optimized version of the LZO algorithm, sometimes surpassing the SPARC 168 MHz.

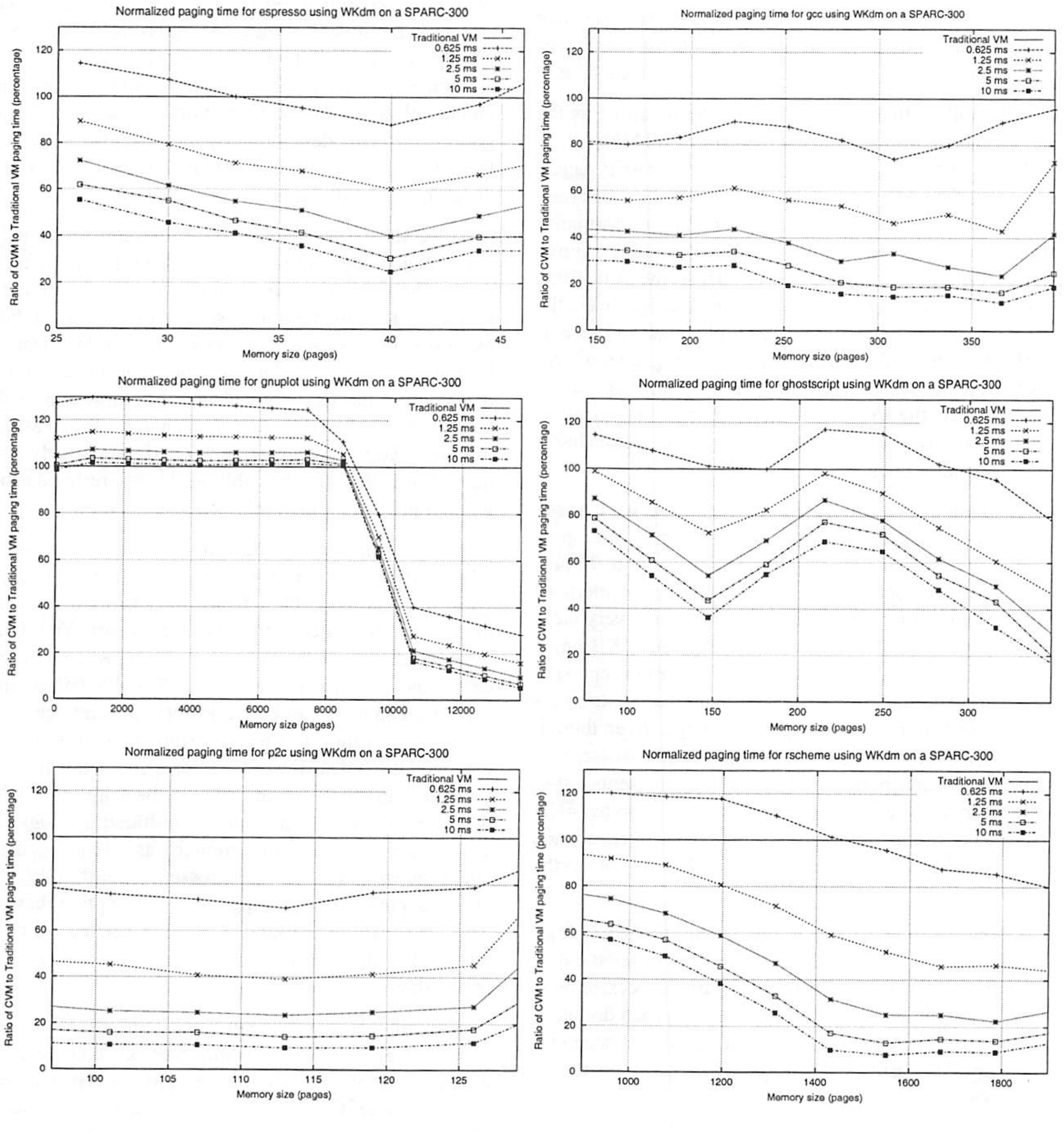


Figure 6: A sensitivity analysis studying disks of various speeds. This conservatively covers the cases of slower CPUs, perfect prefetching, and larger page sizes.

Figure 6 shows plots of simulated performance of our adaptive caching system, using page compression timings measured on a 300 MHz UltraSPARC. Each line represents the paging costs for simulations using a given disk fault cost. Costs are normalized to the performance of a conventional LRU memory with the *same* disk page access time; that is, each curve represents the speedup or slowdown that comes from using compressed caching.

The middle line in each plot can be regarded as the performance of a machine the speed of a 300 MHz UltraSPARC with an average page fetch cost (for 4KB pages) of only 2.5ms, about one third the average disk seek time of a fast disk. Note that, in normalized performance terms, assuming a twice as fast disk is exactly equivalent to assuming a twice as slow CPU. At the same time, studying the case of a fast disk conservatively covers the case of perfect prefetching of multiple pages (a twice as fast disk is equivalent to always prefetching the next two needed pages with one seek). This, in turn, conservatively covers the case of using larger page sizes. Hence, our sensitivity analysis (taking into account fast disks) also subsumes many other scenarios.

Looking at the middle line of each plot, we can see that with a disk page access cost of 2.5ms, most programs show a reduction of paging times by 30 to 70 percent, averaged across the interesting range of memory sizes. Thus, compressed virtual memory is a very clear win even for a disk access cost of 2.5ms per 4KB page. The line above the middle one can be taken to represent a system with the same CPU speed and disk costs a factor of two lower, at 1.25ms per 4KB page. Even though performance for this system is significantly worse, still much speedup is obtained. The top line represents a system where disk page accesses cost only 0.625ms per 4KB page. For some programs, this degrades performance overall to the point that compressed caching is not worthwhile.

Going the other direction, along with the technology trends, we can look at the next lower line to see the performance of a system with twice as fast a processor relative to its disk. For most of our programs, each doubling of CPU speed offers a significant additional speedup, typically decreasing remaining paging costs by ten to forty percent.

5 Related Work

Our compression algorithms are roughly similar to the well-known MTF ("move-to-front") algorithm, which maintains an LRU ordering, but is unusual in its use of partial matching and a fixed 32-bit word as its basic granularity of operation. (The general MTF scheme is fairly obvious and has been invented independently at least four times [BCW90] before we reinvented it yet again.)

The use of partial matching (only the high bits) can be viewed as a simple and fast approximation of *delta coding*, a technique used for purely numeric data (such as sensor input data or digitized audio) [Nel95].[4] Delta coding (a form of differential coding) encodes a numerical value as a numerical difference from the previous numerical value. Unlike a traditional delta coder, our algorithm can encode a value by its difference (low bits) from any of the values in an MTF dictionary, rather than the unique previous value.

In [KGJ96], Kjelso, Gooch, and Jones presented a compression algorithm also designed for in-memory data. Their X-match algorithm (which is designed for hardware implementation) is similar to ours in that both use a small dictionary of recently used words. Rizzo, in [Riz97], also devised a compression algorithm specific to in-memory data. His approach was to compress away the large number of zeros found in such data. Rizzo asserts that more complex modeling would be too costly. We have shown that it is possible to find more regularities without great computational expense.

While we have not addressed the compression of machine code, others have shown that it is possible to compress machine code by a factor of 3 using a specially tuned version of a conventional compressor [Yu96] and by as much as a factor of 5 using a compressor that understands the instruction set [EEF^{+}97]. We believe that similar techniques can be made very fast and achieve a compression ratio of at least 2, similar to the ratios we get for data, so an overall compression ratio of 2 for both code and data should generally be achievable. This is within 20% of the size reduction found by Cogswell and Russinovich using an extremely fast, simple, and untuned "general purpose" compression algorithm [RC96]. (Their paging data also support the assumption that full workloads exhibit the kind of locality needed for compressed paging, making our focus on data paging more reasonable.)

A significant previous study of compressed caching was done by Douglis, who implemented a compressed virtual memory for the Sprite operating system and evaluated it on a DECStation 5000, which is several times to an order of magnitude slower than the machines we used in our experiments.

Douglis's results were mixed, in that compressed virtual memory was beneficial for some programs and detrimental to others. As should be apparent from our dis-

[4]"Delta coding" is something of a misnomer because it's really a modeling technique with an obvious encoding strategy.

cussion of performance modeling, we believe that this was primarily due to the slow hardware (by today's standards) used. This is supported by our sensitivity analysis, which showed that an 8 times slower machine than a 300 MHz UltraSPARC would yield mixed results, even with better compression algorithms than those available to Douglis.

As discussed earlier, Russinovich and Cogswell's study [RC96] showed that a simple compression cache was unlikely to achieve significant benefits for the PC application workload they studied. Nevertheless, their results do not seem to accurately reflect the trade-offs involved. On one hand, they reported compression overheads that seem unrealistically low (0.05ms per compression on an Intel 80486 DX2/66, which is improbable even taking only the memory bandwidth limitations into account). But the single factor responsible for their results is the very high overhead for handling a page fault that they incurred (2ms—this is overhead not containing the actual seek time). This overhead is certainly a result of using a slow processor but it is possibly also an artifact of the OS used (Windows 95) and their implementation.

A study on compressed caching, performed in 1997 but only very recently published, was done by Kjelso, Gooch, and Jones [KGJ99]. They, too, used simulations to demonstrate the efficacy of compressed caching. Additionally, they addressed the problem of memory management for the variable-size compressed pages. Their experiments used the LZRW1 compression algorithm in software and showed for most programs the same kinds of reduction in paging costs that we observed. These benefits become even greater with a hardware implementation of their X-match algorithm.

Kjelso, Gooch, and Jones did not, however, address the issue of adaptively resizing the compressed cache in response to reference behavior. Instead, they assumed that it is always beneficial to compress more pages to avoid disk faults. This is clearly not true as when more pages are compressed, many more memory accesses may suffer a decompression overhead, while only a few disk faults may be avoided. The purpose of our adaptive mechanism is to determine when the trade-off is beneficial and compression should actually be performed. Kjelso, Gooch, and Jones did acknowledge that some compressed cache sizes can damage performance. Indeed, their results strongly suggest the need for adaptivity: two of their four test programs exhibit performance deterioration under software compression for several memory sizes.

6 Conclusions

Compressed virtual memory appears quite attractive on current machines, offering an improvement of tens of percent in virtual memory system performance. This improvement is largely due to increases in CPU speeds relative to disk speeds, but substantial additional gains come from better compression algorithms and successful adaptivity to program behavior.

For all of the programs we examined, on currently available hardware, a virtual memory system that uses compressed caching will incur significantly less paging cost. Given memory sizes for which running a program suffers tolerable amounts of paging, compressed caching often eliminates 20% to 80% of the paging cost, with an average savings of approximately 40%. As the gap between processor speed and disk speed increases, the benefit will continue to improve.

The recency based approach to adaptively resizing the compression cache provides substantial benefit at nearly any memory size, for many kinds of programs. In our tests, the adaptive resizing provided benefit over a very wide range of memory sizes, even when the program was paging little. The adaptivity is not perfect, as small cost may be incurred due to failed attempts to resize the cache, but performs well for the vast majority of programs. Moreover, it is capable of providing benefit for small, medium, and large footprint programs.

The WK compression algorithms successfully take advantage of the regularities of in-memory data, providing reasonable compression at high speeds. After many decades of development of Ziv-Lempel compression techniques, our WKdm compressor fared favorably with the fastest known LZ compressors. Further research into in-memory data regularities promises to provide tighter compression at comparable speeds, improving the performance and applicability of compressed caching for more programs.

It appears that compressed caching is an idea whose time has come. Hardware trends favor further improvement in compressed caching performance. Although past experiments failed to produce positive results, we have improved on the components required for compressed caching and have found that it could be successfully applied today.

References

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Fourth International Conference on Architectural Support for Programming Languages and*

Operating Systems (ASPLOS IV), pages 96–107, Santa Clara, California, April 1991.

[BCW90] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Dou93] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of 1993 Winter USENIX Conference*, pages 519–529, San Diego, California, January 1993.

[EEF^{+}97] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. In *Proceedings of the 1997 SIGPLAN Conference on Programming Language Design and Implementation*, Las Vega, Nevada, June 1997. ACM Press.

[KGJ96] Morten Kjelso, M. Gooch, and S. Jones. Main memory hardware data compression. In *22nd Euromicro Conference*, pages 423–430. IEEE Computer Society Press, September 1996.

[KGJ99] M. Kjelso, M. Gooch, and S. Jones. Performance evaluation of computer architectures with main memory data compression. In *Journal of Systems Architecture 45*, pages 571–590. Elsevier Science, 1999.

[Nel95] Mark Nelson. *The Data Compression Book (2nd ed.)*. M & T Books, 1995.

[RC96] Mark Russinovich and Bryce Cogswell. RAM compression analysis, February 1996. O'Reilly Online Publishing Report available from http://ftp.uni-mannheim.de/info/OReilly/windows/win95.update/model.html.

[Riz97] Luigi Rizzo. A very fast algorithm for RAM compression. In *Operating Systems Review 311997*, pages 36–45, 1997.

[SW91] Walter R. Smith and Robert V. Welland. A model for address-oriented software and hardware. In *25th Hawaii International Conference on Systems Sciences*, January 1991.

[Wil90] Paul R. Wilson. Some issues and strategies in heap management and memory hierarchies. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, October 1990. Also appears in *SIGPLAN Notices 23*(3):45–52, March 1991.

[Wil91a] Ross N. Williams. An extremely fast Ziv-Lempel compression algorithm. In *Data Compression Conference*, pages 362–371, April 1991.

[Wil91b] Paul R. Wilson. Operating system support for small objects. In *International Workshop on Object Orientation in Operating Systems*, pages 80–86, Palo Alto, California, October 1991. IEEE Press.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

[WKB] Paul R. Wilson, Scott F. Kaplan, and V.B. Balayoghan. Virtual memory reference tracing using user-level access protections. In Preparation.

[WLM91] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–191, Toronto, Ontario, June 1991. ACM Press. Published as *SIGPLAN Notices 26*(6), June 1992.

[Yu96] Tong Lai Yu. Data compression for PC software distribution. *Software Practice and Experience*, 26(11):1181–1195, November 1996.

The UVM Virtual Memory System

Charles D. Cranor Gurudatta M. Parulkar

Department of Computer Science
Washington University
St. Louis, MO 63130
{chuck,guru}@arl.wustl.edu

Abstract

We introduce UVM, a new virtual memory system for the BSD kernel that has an improved design that increases system performance over the old Mach-based 4.4BSD VM system. In this paper we present an overview of both UVM and the BSD VM system. We focus our discussion on the design decisions made when creating UVM and contrast the UVM design with the less efficient BSD VM design. Topics covered include mapping, memory object management, anonymous memory and copy-on-write mechanisms, and pager design. We also present an overview of virtual memory based data movement mechanisms that have been introduced in BSD by UVM. We believe that the lessons we learned from designing and implementing UVM can be applied to other kernels and large software systems. Implemented in the NetBSD operating system, UVM will completely replace BSD VM in NetBSD 1.4.

1 Introduction

In this paper we introduce UVM[1], a new virtual memory system that replaces the 4.4BSD virtual memory system in BSD-based operating systems. UVM is the third generation BSD virtual memory system that improves the performance and reduces the complexity of the BSD kernel. UVM's improved performance is particularly useful to applications that make heavy use of VM features such as memory-mapped files and copy-on-write memory.

Versions of BSD prior to 4.4BSD used the old BSD-VAX virtual memory system that was tightly bound to the VAX architecture and lacked support for memory-mapped files (`mmap`) and fine-grain data structure locking for multiprocessors. To address these issues, the old VAX-based VM system was replaced with a new VM system for 4.4BSD [12]. The 4.4BSD virtual memory system (BSD VM) is a modified version of the virtual memory system that was written for Carnegie Mellon University's Mach operating system [18]. The BSD VM system features a clean separation of machine-dependent functions, support for `mmap`, and fine-grain data structure locking suitable for multiprocessor systems.

[1]Note that "UVM" is a name, not an acronym

1.1 Why Replace BSD VM?

The BSD VM system has four main drawbacks that contributed to our decision to replace it: complex data structures, poor performance, no virtual memory based data movement mechanisms, and poor documentation.

The data structures and functions used by BSD to manage memory are complex and thus difficult to maintain. This is especially true of the structures used to represent copy-on-write mappings of memory objects. As memory objects are copied using the copy-on-write mechanism [2] (e.g., during a fork) they are linked together in lists called *object chains*. If left unchecked, use of the copy-on-write mechanism can cause object chains to grow quite long. Long object chains are a problem for two reasons. First, long object chains slow memory search times by increasing the number of objects that need to be checked to locate a requested page. Second, long object chains are likely to contain inaccessible redundant copies of the same page of data, thus wasting memory. If the BSD VM system allows too many pages of memory to be wasted this way the system's swap area will eventually become filled with redundant data and the system will deadlock. This condition is known as a *swap memory leak* deadlock. To avoid problems associated with long object chains, the BSD VM system attempts to reduce their length by using a complex *collapse* operation. To successfully collapse an object chain, the VM system must search for an object that contains redundant data and is no longer needed in the chain. If a redundant object is found, then it is either bypassed or discarded. Note that even a successfully collapsed object chain can still contain inaccessible redundant pages. The collapse operation can only repair swap memory leaks after they occur, it cannot prevent them from happening.

BSD VM exhibits poor performance due to several factors. First, the overhead of object chain management slows down common operations such as page faults and memory mapping. Second, I/O operations in BSD VM are performed one page at a time rather than in more efficient multi-page clusters, thus slowing paging response time. Third, BSD VM's integration into the BSD kernel has not been optimized. For example, unreferenced memory-mapped file objects are cached at the I/O system (vnode) layer and redundantly at the virtual memory layer as well. Finally, several virtual memory operations are unnecessarily performed multiple times at different layers of the BSD kernel.

Another drawback of the BSD VM system is its lack of virtual memory based data movement mechanisms. While BSD VM does support the copy-on-write mechanism, it is not possible in BSD VM for the virtual memory system to safely share memory it controls with other kernel subsystems such as I/O and IPC without performing a costly data copy. It is also not possible for processes to easily exchange, copy, or share chunks of their virtual address space between themselves.

Finally, the BSD VM system is poorly documented. While some parts of the BSD kernel such as the networking and IPC system are well documented [20], the BSD VM system lacks such detailed documentation and is poorly commented. This has made it difficult for developers of free operating systems projects such as NetBSD [17] to understand and maintain the 4.4BSD VM code.

1.2 The UVM Approach

One way to address the shortcomings of the BSD virtual memory system is to try and evolve the data structures and functions BSD inherited from Mach into a more efficient VM system. This technique has been successfully applied by the FreeBSD project to the BSD VM system. However, the improved VM system in FreeBSD still suffers from the object chaining model it inherited from BSD VM. An alternative approach is to reimplement the virtual memory system, reusing the positive aspects of the BSD VM design, replacing the parts of the design that do not work well, and adding new features on top of the resulting VM system. This is the approach we used for UVM. In UVM we retained the machine-dependent/machine-independent layering and mapping structures of the BSD VM system. We replaced the virtual memory object, fault handling, and pager code. And, we introduced new virtual memory based data movement mechanisms into UVM. When combined with I/O and IPC system changes currently under development, these mechanisms can reduce the kernel's data copying overhead.

UVM's source code is freely available under the standard BSD license. UVM was merged into the NetBSD operating system's master source repository in early 1998 and has proven stable on the architectures supported by NetBSD including the Alpha, I386, M68K, MIPS, Sparc, and VAX. UVM will appear as the official virtual memory system of NetBSD in NetBSD release 1.4. A port of UVM to OpenBSD is also in progress.

In this paper we present the design and implementation of the UVM virtual memory system, highlighting its improvements over the BSD VM design. In Section 2 we present an overview of BSD VM and UVM. In Sections 3, 4, 5, 6, and 7 we present the design of UVM's map, object, anonymous memory, pager, and data movement facilities, respectively. In Section 8 we present the overall performance of UVM. Section 9 contains related work, and Section 10 contains our concluding remarks and directions for future research.

2 VM Overview

Both BSD VM and UVM can be divided into two layers: a small machine-dependent layer, and a larger machine-independent layer. The machine-dependent layer used by both BSD and UVM is called the *pmap* layer. The pmap layer handles the low level details of programming a processor's MMU. This task consists of adding, removing, modifying, and querying the mappings of a virtual address or of a page of physical memory. The pmap layer has no knowledge of higher-level operating system abstractions such as files. Each architecture supported by the BSD kernel must have its own pmap module. Note that UVM was designed to use the same machine-dependent layer that BSD VM and Mach use. This allows pmap modules from those systems to be reused with UVM, thus reducing the overhead of porting a UVM-based kernel to a new architecture.

The machine-independent code is shared by all BSD-supported processors and contains functions that perform the high-level operations of the VM system. Such functions include managing a process' file mappings, requesting data from backing store, paging out memory when it becomes scarce, managing the allocation of physical memory, and managing copy-on-write memory. Figure 1 shows the five main abstractions that correspond to data structures in both BSD VM and UVM that the activities of the machine-independent layer are centered around. These abstractions are:

virtual memory space: describes both the machine-dependent and machine-independent parts of a pro-

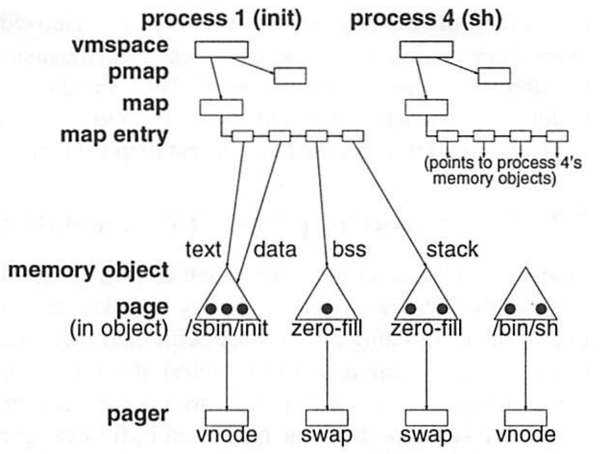


Figure 1: The five main machine-independent abstractions. The triangles represent memory objects, and the solid circles within them represent pages. A memory object can contain any number of pages. Note that the text and data areas of a file are different parts of a single object.

cess' virtual address space. The `vmspace` structure contains pointers to a process' pmap and memory map structures, and contains statistics on the process' memory usage.

memory map: describes the machine-independent part of the virtual address space of a process or the kernel. Both BSD VM and UVM use map (`vm_map`) structures to map memory objects into regions of a virtual address space. Each map structure on the system contains a sorted doubly-linked list of *entry* structures. Each entry structure contains a record of a mapping in the map's virtual address space. This record includes information such as starting and ending virtual address and a pointer to the memory object mapped into that address range. The entry also contains the attributes of the mapping. Attributes include protection, memory usage pattern (advice), and wire count. The kernel and each process on the system have their own map structures to handle the allocations of their virtual address space.

memory object: describes a file, a zero-fill memory area, or a device that can be mapped into a virtual address space. Memory objects contain a list of pages that contain data from that object. In BSD VM, a memory object consists of one or more `vm_object` structures. In UVM, a memory object consists of either a `vm_amap` anonymous memory map structure or a `uvm_object` structure (or both). The details of the handling of memory objects in both BSD VM and UVM are discussed in Section 4 and Section 5.

pager: describes how backing store can be accessed. Each memory object on the system has a pager that points to a list of functions used by the object to fetch and store pages between physical memory and backing store. Pages are read in from backing store when a process faults on them (or in anticipation of a process faulting on them). Pages are written out to backing store at the request of a user (e.g., via the `msync` system call), when physical memory is scarce, or when the object that owns the pages is freed.

page: describes a page of physical memory. When the system is booted a `vm_page` structure is allocated for each page of physical memory that can be used by the VM system[2].

In Figure 1, the system has just been booted single-user so there are only two processes (`init` and `sh`). The `init` process has four entries in its memory map. These entries map the process' text, data, bss, and stack. The entries are sorted by starting virtual address. Each entry describes a mapping of a memory object into `init`'s address space. Note that a single memory object can be mapped into different areas of an address space. For example, the `/sbin/init` file is mapped into `init`'s address space twice, once for the text and once for the data. These mappings must be separate because they have different protections. Each memory object has a list of pages containing its resident data, and a pointer to a pager that can transfer data between an object's pages and backing store. Note that each process' map structure has an associated pmap structure that contains the low-level machine-dependent memory management information (e.g., page tables) for that process' virtual address space.

When a process attempts to access an unmapped area of memory a page fault is generated. The VM system's page fault routine resolves page faults by locating and mapping the faulting page. In order to find which page should be mapped, the VM system must look in the process' map structure for the entry that corresponds to the faulting address. If there is no entry mapping the faulting address an error signal is generated. If an object is mapped at the faulting address, the VM system must determine if the requested data is already resident in a page. If so, that page can be mapped in. If not, then the fault routine must issue a request to the object's pager to make the data resident and resolve the fault.

[2]On a few systems that have small hardware page sizes, such as the VAX, the VM system has a page structure that manages two or more hardware pages. This is all handled in the pmap layer and thus is transparent to the machine-independent VM code.

The following sections examine the design and management of these five abstractions in more detail.

3 Memory Maps

UVM introduces two important improvements to memory maps. First, we have redesigned the memory mapping functions so that they are more efficient and secure. Second, we have greatly reduced map entry fragmentation due to memory wiring.

3.1 Memory Mapping Functions

The `uvm_map` and `uvm_unmap` functions are two of a number of functions that perform operations on maps. The `uvm_map` function is used to establish a new memory mapping with the specified attributes. The `uvm_map` function operates by locking the map, adding the mapping, and then unlocking the map. The BSD VM system does not have an equivalent function to `uvm_map`. Instead, BSD VM provides a function that establishes a mapping with default attributes and a set of functions that change the attributes of a mapping. This is both inefficient and insecure. It is inefficient because it requires extra map locking and lookup steps to establish a mapping with non-default values. For example, the default BSD VM protection is read-write, and thus establishing a read-only mapping under BSD VM is a two step process. First, the mapping must be established with the default protection. Second, the map must be relocked and the desired mapping located again in order to change its protection from read-write to read-only. Note that when establishing a read-only mapping, there is a brief period of time between the first and second step where the mapping has been fully established with a read-write protection. Under a multithreaded kernel two threads sharing the same address space could exploit this window to bypass system security and illegally modify read-only data.

Both BSD VM and UVM have unmap functions with the same API, however the internal structure of these two functions differ. The BSD VM unmap function keeps the target map locked for a longer period of time than necessary, thus blocking other threads from accessing it. In BSD VM, an unmap operation is performed by locking the map, removing the requested map entries, dropping the references to the mapped memory objects, and then unlocking the map. Though the map is locked throughout BSD VM's unmap operation, it really only needs to be locked when removing entries from the map. The target map does not need to be locked to drop references to memory objects (note that dropping the final reference to a memory object can trigger lengthy I/O operations). UVM's unmap function breaks the unmap operation into two phases. In the first phase the target map is locked while the requested map entries are removed. Once this is complete, the map is unlocked and the memory object references can be dropped. The second phase is done with the target map unlocked, thus reducing the total amount of time access to the target map is blocked.

3.2 Wiring and Map Entry Fragmentation

Map entry fragmentation occurs when an area of virtual memory mapped by a single map entry is broken up into two or three adjoining pieces, each with their own map entry. Map entry fragmentation is undesirable for a number of reasons. First, the more entries a map has the longer it takes to perform operations on it, for example, searching the map for the proper entry when resolving a page fault. Second, the process of fragmenting a map entry can add overhead to a mapping operation. To fragment a map entry, new map entries must be allocated and initialized, and additional references to backing objects must be gained. Finally, in the case of the kernel, the total number of available map entries is fixed. If the kernel's pool of map entries is exhausted then the system will fail. While map entry fragmentation is unavoidable in many cases, it is clearly to the kernel's advantage to reduce it as much as possible.

Map entry fragmentation occurs when modifications are made to only part of an area of virtual memory mapped by an entry. Since all pages of virtual memory mapped by a single map entry must have the same attributes, the entry must be fragmented. For example, the adjoining text and data segments of the `init` process shown in Figure 1 must be mapped by separate map entries because they have different protections. Once a map entry has been fragmented neither BSD VM nor UVM will attempt to reassemble it in the unlikely event that the attributes are changed to be compatible.

One of the most frequent causes of map entry fragmentation is the wiring and unwiring of virtual memory. Wired memory is memory that must remain resident in physical memory, and thus cannot be paged out. In BSD, there are five ways for memory to be wired. Unlike BSD VM, UVM avoids map entry fragmentation and the penalties associated with it in four out of five of these cases by taking advantage of the fact that the wired state of a page is often stored in other areas of memory in addition to the entry mapping it, and thus there is no need to disturb the map structure. Memory is wired by the BSD kernel in the following cases:

- The memory used to store the kernel's code segment, data segment, and dynamically allocated data structures is wired to prevent the kernel from taking an unexpected page fault in critical code. Since this memory is always wired, there is no need for UVM to note this fact in the kernel's map structure.

- Each process on the system has a `user` structure containing its kernel stack, signal information, and process control block. A process' `user` structure must be wired as long as the process is runnable. When a process is swapped out its `user` structure is unwired until the process is swapped back in. In this case the wired state of the `user` structure is stored in the process' `proc` structure and there is no need for UVM to store it in the kernel's map as well.
- The `sysctl` system call is used to query the kernel about its status. The `sysctl` call temporarily wires the user's buffer before copying the results of the call to it. This is done to minimize the chances of a page fault causing inconsistent data to be returned. Rather than store the wired state of the user's buffer in the process' map, UVM stores this information on the process' kernel stack since the buffer is only wired while the `sysctl` operation is in progress.
- The kernel function `physio` is used to perform raw I/O between a device and a user process' memory. Like `sysctl`, `physio` temporarily wires the user's buffer while the I/O is in progress to keep it resident in physical memory. And like `sysctl`, UVM stores the wired state of the users' buffer on the process' kernel stack.
- The kernel provides the `mlock` system call to allow processes to wire their memory to avoid page faults in time-critical code. In this case the wired state of memory must be stored in the user process' map because there is no other place to store it.

In addition to these five cases, under the i386 architecture the machine-dependent code uses wired memory to allocate hardware page tables. Under UVM the wired state of page table memory is stored only in the i386's pmap structure rather than in both the pmap structure and the user process' map.

By reducing the amount of map entry fragmentation due to wired memory, we significantly lowered map entry demand under UVM. For example, consider the statically linked program `cat` and the dynamically linked program `od`. On the i386 platform, BSD VM requires 11 map entries for `cat` and 21 for `od`, while UVM requires only six map entries for `cat` and 12 for `od`. The difference between BSD VM and UVM is due to the `user` structure allocation, the `sysctl` system call, and the i386's pmap page table allocation routine. We found that calls to `mlock` and `physio` seldom occur under normal system operation. Table 1 shows a comparison of the number of allocated map entries for several common operations. While the effect of this reduction in the number of allocated map entries on overall system performance is minimal, it should be noted that the total number of map entries available for the kernel is fixed and if this pool is exhausted the system will panic. This could become a problem under BSD VM since each process requires two kernel map entries.

Operation	**Number of Map Entries**	
	BSD	**UVM**
`cat` (static link)	11	6
`od` (dynamic link)	21	12
single-user boot	50	26
multi-user boot (no logins)	400	242
starting X11 (9 processes)	275	186

Table 1: Comparison of the number of allocated map entries on the i386 for some common operations. On an i386 a map entry is fifty-six bytes.

4 Memory Objects

UVM manages memory objects significantly differently than BSD VM. In BSD VM, the memory object structure is considered a stand-alone abstraction under the control of the VM system. BSD VM controls when objects are allocated, when they can be referenced, and how they can be used. In contrast, in UVM the memory object structure is considered a secondary structure designed to be embedded within some larger structure in order to provide UVM with a handle for memory mapping. The structure in which UVM's memory object is embedded is typically part of a structure managed externally to the VM system by some other kernel subsystem. For example, UVM's object structure for file data is embedded within the I/O system's vnode structure. The vnode system handles the allocation of UVM's memory object structure along with the allocation of the vnode structures. All access to the memory object's data and state is routed through the object's pager functions. These functions act as bridge between UVM and the external kernel subsystem that provides UVM with its data (see Section 6).

UVM's style of management of memory objects is preferable to BSD VM's style for several reasons. First, UVM's management of memory objects is more efficient than BSD VM's. In UVM, memory objects are allocated and managed in cooperation with their data source (typically vnodes). In BSD VM, memory objects and their data sources must be allocated and managed separately. This causes the BSD VM system to duplicate work that the data source subsystem has already performed. BSD VM must allocate more structures and have more object management code than UVM to perform the same operations.

Second, UVM's memory object structure is more flexible than BSD VM's structure. By making the memory object an embeddable data structure, it is easy to make any kernel abstraction memory mappable. Additionally, UVM's routing of object requests through its pager operations gives the external kernel subsystem that generates the memory object's data a finer grain of control over how UVM uses it.

Finally, UVM's memory object management structure creates less conflict between the VM system and external kernel subsystems such as the vnode subsystem. BSD's vnode subsystem caches unreferenced vnodes in physical memory in hopes that they will be accessed again. If vnodes become scarce, then the kernel recycles the least recently used unreferenced vnode. In the same way, the BSD VM system caches unreferenced memory objects. While vnode structures are allocated when a file is opened, read, written, or memory mapped, BSD VM vnode-based memory objects are allocated only when a file is memory mapped. When an unreferenced memory object is persisting in BSD VM's object cache, the VM system gains a reference to the object's backing vnode to prevent it from being recycled out from under it. Unfortunately, this also means that there are times when the most optimal unreferenced vnode to recycle is in BSD VM's object cache, resulting in the vnode system choosing a non-optimal vnode to recycle. Another problem with the BSD VM object cache is that it is limited to one hundred unreferenced objects in order to prevent the VM system from holding too many active references to vnode structures (preventing recycling). If the BSD VM system wants to add an unreferenced object to a full cache, then the least recently used object is discarded. This is less than optimal because the object's vnode data may still be persisting in the vnode system's cache and it would be more efficient to allow the memory object to persist as long as its vnode does.

Rather than having two layers of unreferenced object caching, UVM has only one. Instead of maintaining its own cache, UVM relies on external kernel subsystems such as the vnode system to manage the unreferenced object cache. This reduces redundant code and allows the least recently used caching mechanism to be fairly applied to both vnodes and memory objects. When recycling a vnode, UVM provides the vnode subsystem with a hook to terminate the memory object associated with it. This change can have a significant effect on performance. For example, consider a web server such as Apache that transmits files by memory mapping them and writing them out to the network. If the number of files in the server's working set is below the one-hundred-file limit, then both BSD VM and UVM can keep all the file data resident in memory. However, if the working set grows beyond one hundred files, then BSD VM flushes older inactive objects out of the object cache (even if memory is available). This results in BSD VM being slowed by disk access. Figure 2 shows this effect measured on a 333MHz Pentium-II. To produce the plot we wrote a program that accesses files in the same way as Apache and timed how long it took to memory map and access each byte of an increasing number of files.

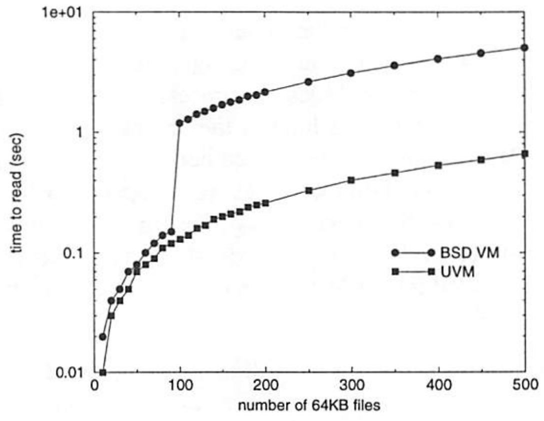


Figure 2: BSD VM object cache effect on file access

5 Anonymous Memory

Anonymous memory is memory that is freed as soon as it is no longer referenced. This memory is referred to as *anonymous* because it is not associated with a file and thus does not have a file name. Anonymous memory is paged out to the swap area when memory is scarce. Anonymous memory is used for a number of purposes in a Unix-like operating system including for zero-fill mappings (e.g., bss and stack), for System V shared memory, for pageable areas of kernel memory, and to store changed pages of a copy-on-write mapping. A significant part of the code used to manage anonymous memory is dedicated to controlling copy-on-write memory. In this section we first present a brief overview of the management of anonymous memory in both BSD VM and UVM. We then describe the improvements introduced in UVM which result in the elimination of swap memory leaks, a more efficient copy-on-write mechanism, and less complex code.

5.1 BSD VM Anonymous Memory

Creating an anonymous zero-fill mapping under BSD VM is a straight forward process. BSD VM simply allocates an anonymous memory object of the specified size and inserts a map entry pointing to that object into a map. On the other hand, the management of copy-on-write memory under BSD is more complex.

The BSD VM system manages copy-on-write mappings of memory objects by using *shadow objects*. A shadow object is an anonymous memory object that contains the modified pages of a copy-on-write mapped memory object. The map entry mapping a copy-on-write area of memory points to the shadow object allocated for it. Shadow objects point to the object they are shadowing. When searching for pages in a copy-on-write mapping, the shadow object pointed to by the map entry is searched first. If the desired page is not present in the shadow object, then the underlying object is searched. The underlying object may either be a file object or another shadow object. The search continues until the desired page is found, or there are no more underlying objects. The list of objects that connect a copy-on-write map entry to the bottom-most object is called a *shadow object chain.*

The upper row of Figure 3 shows how shadow object chains are formed in BSD VM. In the figure, a three-page file object is copy-on-write memory mapped into a process' address space. The first column of the figure shows how copy-on-write mappings are established. To establish a copy-on-write mapping the BSD VM system allocates a new map entry, sets the entry's needs-copy and copy-on-write flags, points the map entry at the underlying object (usually a file object), and inserts it into the target map. The needs-copy flag is used to defer allocating a new memory object until the first write fault on the mapping occurs. Once a write fault occurs, a new memory object is created and that object tracks all the pages that have been copied and modified due to write faults. Under BSD VM, needs-copy indicates that the mapping requires a shadow object the next time the mapped memory is modified. Read access to the mapping will cause the underlying object's pages to be mapped read-only into the target map.

The second column in Figure 3 shows what happens when the process writes to the middle page of the object. Since the middle page is either unmapped or mapped read-only, writing to it triggers a page fault. The VM system's page fault routine must catch and resolve this fault so that process execution can continue. The fault routine looks up the appropriate map entry and notes that it is a needs-copy copy-on-write mapping. It first clears needs-copy by allocating a shadow object and inserting it between the map entry and the underlying file. Then it copies the data from the middle page of the backing object into a new page that is inserted into the shadow object. The shadow object's page can then be mapped read-write into the faulting process' address space. Note that the shadow object only contains the middle page. Other pages will be copied only if they are modified.

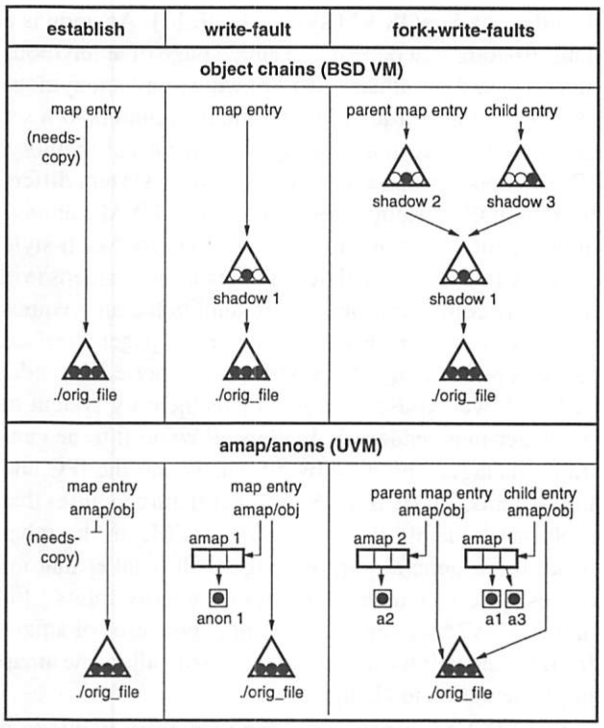


Figure 3: The copy-on-write mechanisms of BSD VM and UVM. In the figures a process establishes a copy-on-write mapping to a three page file (the solid black circles represent pages). When the mapping is established the *needs-copy* flag is set. After the first write fault the needs-copy flag is cleared by allocating a shadow object or an amap. If the process forks and more write faults occur additional shadow objects and amaps are allocated.

The third column in Figure 3 shows the BSD VM data structures after the process with the copy-on-write mapping forks a child, the parent writes to the middle page, and the child writes to the right-hand page. When the parent forks, the child receives a copy-on-write copy of the parent's mapping. This is done by write protecting the parent's mappings and setting needs-copy in both processes. When the parent faults on the middle page, a second shadow object is allocated for it (clearing needs-copy) and inserted on top of the first shadow object. When the child faults on the right-hand page the same thing happens, resulting in the allocation of a third shadow object.

5.2 UVM Anonymous Memory

UVM manages anonymous memory using an extended version of the *anon* and *amap* abstractions first introduced in the SunOS VM system [4, 9, 13]. An anon is a data structure that describes a single page of anonymous memory, and an amap (also known as an "anonymous map") is a data structure that contains pointers to a set of anons that are mapped together in virtual memory. UVM's amap-based anonymous memory system differs from SunOS' system in four ways. First, UVM's anonymous memory system introduces support for Mach-style memory inheritance and deferred creation of amaps (via the needs-copy flag). Second, in SunOS the anonymous memory system resides below the vnode pager interface and was not designed to be visible to generic VM code. In UVM, we expose the anonymous memory system to the pager-independent code, thus allowing it to be centrally managed and used by all pagers and the IPC and I/O systems. Third, SunOS' pager structure requires that each pager handle its own faults. UVM, on the other hand, has a general purpose page fault handler that includes code to handle anonymous memory faults. Finally, in UVM we separate the implementation of amaps from the amap interface in order to easily allow the amap implementation to change.

In BSD VM, a copy-on-write map entry points to a chain of shadow objects. There is no limit on the number of objects that can reside in a single shadow object chain. UVM, on the other hand, uses a simple two-level mapping scheme consisting of an upper amap anonymous memory layer and a lower backing object layer. In UVM, a copy-on-write map entry has pointers to the amap and underlying object mapped by that entry. Either pointer can be null. For example, a shared mapping usually has a null amap pointer and a zero-fill mapping has a null object pointer.

UVM's anon structure contains a reference counter and the current location of the data (i.e., in memory or on backing store). An anon with a single reference is considered writable, while anons referenced by more than one amap are copy-on-write. To resolve a copy-on-write fault on an anon, the data is copied to a newly allocated anon and the reference to the original anon is dropped. The lower row of Figure 3 shows how UVM handles copy-on-write mappings using the same example used for BSD VM. In UVM a copy-on-write mapping is established by inserting a needs-copy copy-on-write map entry pointing to the underlying object in the target map. When the process with the copy-on-write mapping writes to the middle page the UVM fault routine resolves the fault by first allocating a new amap to clear needs-copy and then copying the data from the backing object into a newly allocated anon. The anon is inserted into the middle slot of the mapping's amap.

The third column in the UVM row of Figure 3 shows the UVM data structures after the process with the copy-on-write mapping forks a child process, the parent process writes to the middle page, and the child process writes to the right-hand page. When the parent process forks, the child receives a copy-on-write copy of the parent's mapping. This is done by write protecting the parent's mappings and setting needs-copy in both the parent and child. When the parent process faults on the middle page, a second amap is allocated for it (clearing needs-copy and incrementing the reference count of anon 1) and the data is copied from the first anon (still in the original amap) to a newly allocated anon that gets installed in the new amap. When the child process faults on the right-hand page the fault routine clears needs-copy without allocating a new amap because the child process holds the only reference to the original amap. The fault routine resolves the child's fault by allocating a third anon and installing it in the child's amap.

5.3 Anonymous Memory Comparison

Both BSD VM and UVM use needs-copy to defer the allocation of anonymous memory structures until the first copy-on-write fault. Thus, in a typical fork operation where the child process immediately executes another program most amap copying and shadow object creation is avoided[3]. In both systems there is a per-page overhead involved in write protecting the parent process' mappings to trigger the appropriate copy-on-write faults. To clear needs-copy under UVM a new amap must be allocated and initialized with anon pointers (adding a reference to each anon's reference counter). To clear needs-copy under BSD VM a new shadow object must be allocated and inserted in the object chain. Future write faults require BSD VM to search underlying objects in the chain for data and promote that data to the top-level shadow object. Also, in addition to normal write-fault processing, BSD VM attempts an object collapse operation each time a copy-on-write fault occurs.

BSD VM's kernel data structure space requirements for copy-on-write consist of a fixed-size shadow object and the pager data structures associated with it. The number of pager data structures varies with the number of virtual pages the object maps. Pages are clustered together into swap blocks that can be anywhere from 32KB to 128KB depending on object size. Each allocated swap block structure contains a pointer to a location on backing store. UVM's kernel data structure space requirements for copy-on-write consist of an amap data structure and the anons associated with it. An amap's size is dictated by the amap implementation being used. UVM currently uses an array-based implementation whose space cost varies with the number of virtual pages covered by the amap. This is expensive for

[3]And even this could be avoided with the vfork system call.

larger sparsely allocated amaps, but the cost could easily be reduced by using a hybrid amap implementation that uses both hash tables and arrays. UVM stores swap location information on a per-page basis in anon structures. UVM must store this information on a per-page basis rather than using BSD VM-like swap blocks because UVM supports the dynamic reassignment of swap location at page-level granularity for fast clustered page out (described in Section 6).

There are a number of design problems and shortcomings in BSD VM's anonymous memory system that contributed to our decision to completely replace it with UVM's amap-based anonymous memory system. BSD VM's copy-on-write mechanism can leak memory by allowing pages of memory that are no longer accessible to remain allocated within an object chain. For example, consider the final BSD VM diagram in Figure 3. If the child process exits, then the third shadow object will be freed. The remaining shadow object chain contains three copies of the middle page. Of these three copies only two are accessible — the page in the first shadow object is no longer accessible and should be freed. Likewise, if the child process writes to the middle page rather than exits, then the page in the first shadow object also becomes inaccessible. If such leaks were left unchecked, the system would exhaust its swap space.

Clearly the longer a shadow object chain is, the greater the chance for swap space to be wasted. Although BSD VM cannot prevent shadow object chains from forming, it attempts to reduce the length of a chain after it has formed by collapsing it. BSD VM attempts to collapse a shadow object chain when ever a write fault occurs on a shadow object, a shadow object reference is dropped, a shadow object is copied, or a shadow object pages out to swap for the first time. This work is done in addition to normal VM processing.

Searching for objects that can be collapsed is a complex process that adds extra overhead to BSD VM. To contrast, no collapsing is necessary with UVM because the amap and anon reference counters keep track of when pages should be freed. This allows new features of UVM such as copy-on-write based data movement mechanisms to be implemented more efficiently than under BSD VM.

Another problem with BSD VM's copy-on-write mechanism is that it is inefficient. For example, consider what happens if the child process in Figure 3 writes to the middle page. Under BSD VM, the data in the middle page of shadow object 1 is copied into a new page of shadow object 3 to resolve the fault. This page allocation and data copy are unnecessary. Ideally, rather than copying the data from shadow object 1 to shadow object 3 the middle page from shadow object 1 would simply be reassigned to shadow object 3. Unfortunately this is not possible under BSD VM because the data structure do not indicate if shadow object 1 still needs its page or not. In UVM, writing to the middle page is handled by allowing the child process to directly write to the page in anon 1 (this is allowable because anon 1's reference count is one), thus avoiding the expensive and unnecessary page allocation and data copy.

Finally, the code used to manage anonymous memory under BSD VM is more complex than UVM's amap-based code. BSD VM must be prepared to loop through a multi-level object chain to find needed data. Each object in the chain has its own set of I/O operations, its own lock, its own shadow object, and its own pool of physical memory and swap space. BSD VM must carefully manage all aspects of each object in the chain so that memory remains in a consistent state. At the same time, it needs to aggressively collapse and bypass shadow objects to prevent memory leaks and keep the object chains from becoming too long, thus slowing memory searches. In contrast, UVM can perform the same function using its simple two-level lookup mechanism. Rather than looping through a chain of objects to find data, UVM need only check the amap and then the object layer to find data. Rather than using lists of objects, UVM uses reference counters in amaps and anons to track access to anonymous memory. UVM's new anonymous memory management system has contributed to a noticeable improvement in overall system performance (see Section 8).

5.4 Amap Adaptation Issues

UVM's amap-based anonymous memory system is modeled on the SunOS VM system vnode segment driver anonymous memory system [9, 13]. (Segment drivers in SunOS perform a similar role to pagers in UVM.) While this system is sufficient for SunOS, it required a number of adaptations and extensions in order to function in a BSD environment and to support UVM's new data movement features (described in Section 7). First, SunOS's anonymous memory mechanism is not a general purpose VM abstraction. Instead, it is implemented as a part of the SunOS vnode segment driver. This is adequate for SunOS because copy-on-write and zero-fill memory can be isolated in the vnode layer. However, in UVM parts of the general purpose VM system such as the fault routine and data movement mechanisms require access to amaps. Thus in UVM we have repositioned the amap system as a general purpose machine-independent virtual memory abstraction. This allows any type of mapping to have an anonymous layer.

Second, the BSD kernel uses several mechanisms that are not present in SunOS. In order for UVM to replace BSD VM without loss of function the design of UVM's amap system must account for these mechanisms. For example, BSD supports the `minherit` system call. This system call allows a process to control its children's access to its virtual memory. In traditional Unix-like systems (including SunOS) child processes get shared access to a parent's shared mappings and copy-on-write access to the rest of the mappings. In BSD the traditional behavior is the default, however the `minherit` system call can be used to change this. The `minherit` system call allows a process to designate the inheritance of its memory as "none," "shared," or "copy." This creates cases such as a child process sharing a copy-on-write mapping with its parent, or a child process receiving a copy-on-write copy of a parent's shared mapping. In addition to `minherit`, BSD also uses a mapping's needs-copy flag to defer the allocation of anonymous memory structures until they are needed. SunOS does not have a needs-copy flag. Thus UVM, unlike SunOS, must be prepared to delay the allocation of amaps using needs-copy until they are actually needed. In order to maintain consistent memory for all processes while supporting both `minherit` and needs-copy, UVM's amap code must carefully control when amaps are created and track when they are shared.

A third area where the adaptation of an amap-based anonymous memory system affected the design of UVM is in the design of UVM's page fault routine. In SunOS, other than the map entry lookup, all of the work of resolving a page fault is left to the segment driver. On the other hand, BSD VM has a general purpose page fault routine that handles all aspects of resolving a page fault other than I/O, including memory allocation, and walking and managing object chains. In fact, the majority of the BSD VM fault routine's code is related to object chain management. Neither of these two styles of fault routine is appropriate for UVM. A SunOS style fault routine forces too much pager-independent work into the pager layer, and as UVM does not use object chaining the BSD VM fault routine is not applicable. Thus, a new fault routine had to be written for UVM from scratch. The UVM fault routine first looks up the faulting address in the faulting map. It then searches the mapping's amap layer to determine if the required data is in there. If not, it then checks the backing object layer for the data. If the data is not there, then an error code is returned. In addition to resolving the current page fault, the UVM fault routine also looks for resident pages that are close to the faulting address and maps them in. The number of pages looked for is controlled by the `madvise` system (the default is to look four pages ahead of the faulting address and three pages behind). This can reduce the number of future page faults. Table 2 shows the results of this change on an i386 for several sample commands. Note that this mechanism only works for resident pages

Command	BSD VM	UVM
`ls /`	59	33
`finger chuck`	128	74
`cc`	1086	590
`man csh`	114	64
`newaliases`	229	127

Table 2: Page fault counts on an i386 obtained through `csh`'s "time" command. The `cc` command was run on a "hello world" program.

and thus has a minimal effect on execution time for these non-fault intensive applications. As part of our future work, we plan to modify UVM to asynchronously page in non-resident pages that appear to be useful.

6 Pagers

UVM introduces three important improvements to pagers. The allocation of pager-related data structures has been made more efficient, the pager API has been made more flexible giving the pager more control over the pages it owns, and aggressive clustering has been introduced into the anonymous memory system.

There is a significant difference between the way the pager-related data structures are organized in BSD VM and UVM. In BSD VM the pager requires several separately allocated data structures. The left side of Figure 4 shows these structures for the vnode pager. In BSD VM a memory object points to a `vm_pager` structure. This structure contains pointers to a set of pager operations and a pointer to a pager-specific private data structure (`vn_pager`). In turn, this structure points to the vnode being mapped. In addition to these structures, BSD VM also maintains a hash table that maps a pager structure to the object it backs (note that there is no pointer from the `vm_pager` to the `vm_object`). To contrast, the right side of Figure 4 shows the UVM pager data structures for a vnode. All VM related vnode data is embedded within the vnode structure rather than allocated separately. The pager data structure has been eliminated—UVM's memory object points directly to the pager operations. So, in order to set up the initial mappings of a file the BSD VM system must allocate three data structures (`vm_object`, `vm_pager`, and `vn_pager`), and enter the pager in the pager hash table. On the other hand, UVM does not have to access a hash table or allocate any data structures. All the data structures UVM needs are embedded within the vnode structure.

Another difference between the BSD VM pager interface and the UVM pager interface is in the API used to fetch data from backing store. To get a page of an

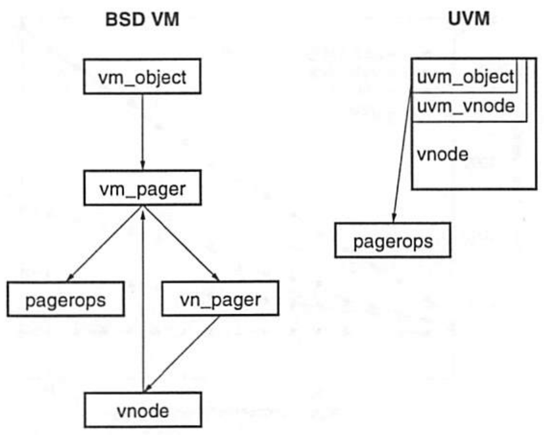


Figure 4: Pager data structures

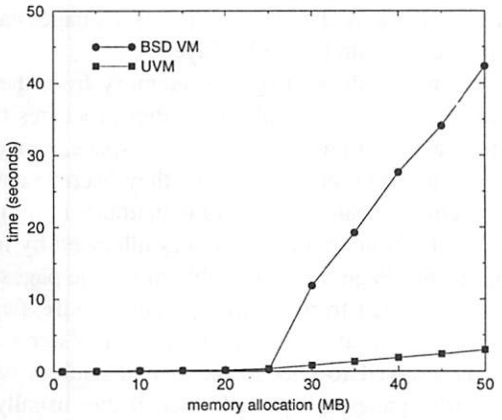


Figure 5: Anonymous memory allocation time under BSD VM and UVM

object's data from backing store in BSD VM, the VM system must allocate a new page, add it to the object, and then request that the pager fill it with data. In UVM, the process fetching the data does not allocate anything, this is left to the pager. If a new page is needed the pager will allocate it itself. This API change allows the pager to have full control over when pages get added to an object. This can be useful in cases where the pager wants to specifically choose which page to put the data in. For example, consider a pager that wants to allow a process to map in code directly from pages in a ROM.

Another difference between the BSD VM pager interface and UVM's pager interface is how UVM handles paging out anonymous memory. One unique property of anonymous memory is that it is completely under the control of the VM system and it has no permanent home on backing store. UVM takes advantage of this property to more aggressively cluster anonymous memory than is possible with the scheme used by BSD VM. The key to this aggressive clustering is that UVM's pagedaemon can reassign an anonymous page's pageout location on backing store. This allows UVM's pagedaemon to collect enough dirty anonymous pages to form a large cluster for pageout. Each page's location on swap is assigned (or reassigned) so that the cluster occupies a contiguous chunk of swap and can be paged out in a single large I/O operation. So, for example if UVM's pagedaemon detects dirty pages at page offsets three, five, and seven in an anonymous object it can still group these pages in a single cluster, while the BSD VM would end up performing three separate I/O operations to pageout the same pages. As a result UVM can recover from page shortages quicker and more efficiently than BSD VM. Figure 5 compares the time it take to allocate anonymous memory under BSD VM and UVM on a 333MHz Pentium-II with thirty-two megabytes of RAM. As the allocation size becomes larger than physical memory, the system must start paging in order to satisfy the request. UVM can clearly page the data much faster than BSD VM.

7 Data Movement

UVM includes three new virtual memory based data movement mechanisms that are more efficient than bulk data copies when transferring large chunks of data [6]. Page loanout allows pages from a process' address space to be borrowed by other processes. Page transfer allows for pages from the kernel or other processes to be inserted into a process' address space easily. Map entry passing allows processes to copy, share, or move chunks of their virtual address space between themselves. We are currently in the process of modifying the kernel's I/O and IPC systems to take advantage of these facilities to reduce data movement overhead.

Page loanout allows a process to safely let a shared copy-on-write copy of its memory be used either by other processes, the I/O system, or the IPC system. The loaned page of memory can come from a memory-mapped file, anonymous memory, or a combination of the two. Pages can be loaned into wired pages for the kernel's I/O system, or they can be loaned as pageable anonymous memory for transfer to another process. Page loanout gracefully preserves copy-on-write in the presence of page faults, pageouts, and memory flushes. It also operates in such a way that it provides access to memory at page-level granularity without fragmenting or disrupting the VM system's higher-level memory mapping data structures. An example of where page loanout can be used is when data is transmitted over a socket. Rather than bulk copy the data from the user's memory to the kernel's memory, the user's pages can be directly shared with the socket layer.

Page transfer allows pages of memory from the I/O system, the IPC system, or from other processes to be inserted easily into a process' address space. Once the pages are inserted into the process they become anonymous memory. Such anonymous memory is indistinguishable from anonymous memory allocated by traditional means. Page transfer is able to handle pages that have been copied from another process' address space using page loanout. Also, if the page transfer mechanism is allowed to choose the virtual address where the inserted pages are placed, then it can usually insert them without fragmenting or disrupting the VM system's higher-level memory mapping data structures. Page transfer can be used by the kernel to place pages from other processes, I/O devices, or the kernel directly into the receiving process' address space without a data copy.

Map entry passing allows processes and the kernel to exchange large chunks of their virtual address spaces using the VM system's higher-level memory mapping data structures. This mechanism can copy, move, or share any range of a virtual address space. This can be a problem for some VM systems because it introduces the possibility of allowing a copy-on-write area of memory to become shared with another process. Because map entry passing operates on high-level mapping structures, the per-page cost of map entry passing is less than page loanout or page transfer, however it can increase map entry fragmentation if used on a small number of pages and it cannot be used to share memory with other kernel subsystems that may access pages with DMA. Map entry passing can be used as a replacement for pipes when transferring large-sized data.

The preliminary measurements of UVM's three data movement mechanisms show that VM-based data movement mechanisms improve performance over data copying when the size of the data being transfered is larger than a page. For example, in our tests, single-page loanouts to the networking subsystem took 26% less time than copying data. Tests involving multi-page loanouts show that page loaning can reduce the processing time further, for example a 256 page loanout took 78% less time than copying data. We are currently in the process of applying these mechanisms to real-life applications to determine their effectiveness.

8 Overall UVM Performance

Replacing the old BSD VM system with UVM has improved both the overall efficiency and overall performance of the BSD kernel. For example, Figure 6 shows the total time it takes for a process with a given amount of dynamically allocated anonymous memory to fork a child process and then wait for that child process to exit under both BSD VM and UVM. Thus, the plot measures critical VM-related tasks such as creating a new address space, copying the parent's mappings into the child process, copy-on-write faulting, and disposing of the child's address space. In the upper two plots, the child process writes to its dynamically allocated memory once and then exits (thus triggering a copy-on-write fault). In the lower plots the child exits without accessing the data. In both cases UVM clearly out performs BSD VM.

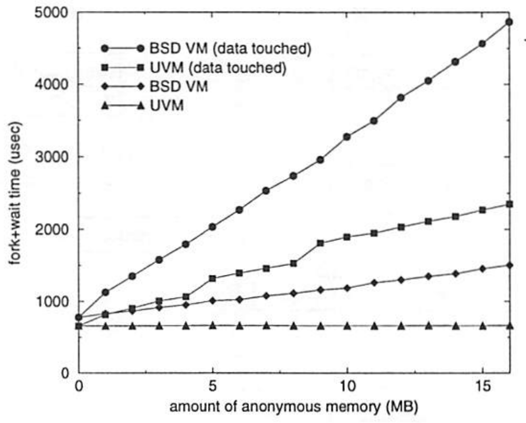


Figure 6: Process fork-and-wait overhead under BSD VM and UVM measured on a 333MHz Pentium-II averaged over 10,000 fork-and-wait cycles

Fault/mapping	**BSD VM** (usec)	**UVM** (usec)
read/shared file	24	21
read/private file	48	22
write/shared file	113	100
write/private file	80	67
read/zero fill	60	49
write/zero fill	60	48

Table 3: Single page map-fault-unmap time on a 333 MHz Pentium II.

Another example of UVM's performance gain is shown in Table 3. The table shows the time (averaged over 1 million cycles) it takes to memory map a page of memory, fault it in, and then unmap the page. UVM outperforms BSD VM in all cases. Note that read faults on a private mapping under BSD VM are more expensive that shared read faults because BSD VM allocates a shadow object for the mapping (even though it is not necessary).

NetBSD users have also reported that UVM's improvements have had a positive effect on their applications. This is most noticeable when physical memory becomes scarce and the VM system must page out data to free up memory. Under BSD VM this type of paging causes the system to become highly unresponsive, while under UVM the system slows while paging but does not become unresponsive. This situation can occur when running large virtual memory intensive applications like a lisp interpreter, or when running a large compile job concurrently with an X server on a system with a small amount of physical memory. In addition to improved responsiveness during paging, users of older architectures supported by NetBSD have noticed that applications run quicker. For example, the running time of `/etc/rc` was reduced by ten percent (ten seconds) on the VAX architecture.

9 Related Work

In UVM, we have focused our efforts on exploring key data structures and mechanisms used for memory management. There has been little recent work in this area, but there has been a lot of work on extensible operating system structure. With UVM, we have created a VM system that is tightly coupled and contains global optimizations that produce a positive impact on system performance. On the other hand, a goal of extensible operating systems is to allow an operating system's functions to be partitioned and extended in user-specified ways. This can be achieved in a number of ways including providing a hardware-like interface to applications (Exokernel [11]), allowing code written in a type safe language to be linked directly into the kernel (SPIN [1]), and allowing software modules to be connected in vertical slices (Scout [14, 19]). While the data structures and mechanisms used by UVM are orthogonal to operating system structure, the effect of extensibility on the tightly coupled global optimizations provided by UVM is unclear. It may be possible to load UVM-like memory management into these systems, for example recent work on the L4 microkernel [10] has shown that a port of Linux to L4 can run with a minimal performance penalty. However, interactions with other extensions may have an adverse effect.

The two virtual memory systems most closely related to UVM are the Mach VM system [18] and the SunOS VM system [4, 9, 13]. Since BSD VM is based on Mach VM, most of the discussion of BSD VM in this paper applies to both VM systems (and to a lesser extent the FreeBSD VM system). As described in Section 5 UVM incorporates and extends parts of SunOS VM's anonymous memory management mechanism. Dyson and Greenman took a different approach to improving the BSD VM data structures in FreeBSD by keeping the same basic structure but eliminating the unnecessary parts of the Mach VM system that BSD inherited [16]. The Linux VM system [21] provides a generic three-level page table based interface to underlying memory management hardware rather than a function-based API like Mach's pmap. All anonymous memory functions are managed through the page table. This is limiting because it does not provide a high-level abstraction for an anonymous page of memory, and it prevents page tables from being recycled when physical memory is scarce. Recent work on virtual memory support for multiple page sizes [8] allows better clustering of I/O operations similar to UVM's aggressive clustering of anonymous memory for page out. However, with large pages data must be copied into a physically contiguous block of memory before it can be paged out. UVM can dynamically reassign anonymous memory's swap location using normal sized pages without copying the data.

Other recent work has focused on zero-copy data movement mechanisms. IO-Lite [15] is a unified buffering system based on Fbufs [7]. IO-Lite achieves zero-copy by forcing all buffers to be immutable once initialized and forcing all I/O operations to be in terms of buffer aggregates. IO-Lite does not interact well with memory-mapped files and is not integrated with a VM system. Solaris zero-copy TCP [5] uses a new low-level pmap API and ATM hardware support to provide zero-copy TCP without effecting higher-level VM code. The L4 microkernel [10] provides granting (remap), mapping, and unmapping primitives to threads to allow for fast VM-based data movement, but it leaves issues such as copy-on-write for higher-level software such as its Linux server. Finally, the Genie I/O subsystem [3] includes mechanisms that allow an operating system to emulate a copy-based API with VM-based mechanisms. Genie's mechanisms could be applied to UVM if such support is desired.

10 Conclusions and Future Work

In this paper we introduced UVM, a new virtual memory system for the BSD kernel. Key aspects of UVM's design include:

- The reuse of BSD VM's machine-dependent layer. This allowed us to focus on the machine-independent aspects of UVM without getting bogged down in machine-specific details. This reuse made porting UVM to architectures supported by BSD VM easy.

- The repartitioning of VM functions in more efficient ways. For example, we combined BSD VM's two-step mapping process into a more efficient and secure single step mapping function, and we broke BSD VM's unmapping function up into smaller functions that hold map locks for a shorter period of time.

- The reduction of duplicate copies of the same state information. This was used in UVM to reduce map entry fragmentation due to page wiring.
- The elimination of the contention between the VM system and other kernel subsystems. For example, we redesigned and improved the management of the inactive memory object cache to work with the vnode system (rather than in parallel to it).
- The elimination of complex data structures such as BSD VM's object chains that make accounting for a program's memory usage difficult and expensive.
- The grouping or clustering of the allocation and use of systems resources to improve system efficiency and performance. For example, we grouped UVM's allocation of pager data structures and we aggressively clustered UVM's anonymous pageout.
- Easy resource sharing with other kernel subsystems. For example, UVM's data movement mechanisms allow it to share its pages with the I/O and IPC subsystem without costly data copies.

Our future plans for UVM include unifying the VM cache with the BSD buffer cache, adding more asynchronous I/O support to UVM (for both pagein and pageout), and adapting the BSD kernel to take advantage of UVM's new data movement mechanisms to improve application performance.

Acknowledgments

We would like to thank Orran Krieger, Lorrie Faith Cranor, and the anonymous reviewers for their helpful comments on drafts of this paper.

References

[1] B. Bershad et al. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth Symposium on Operating System Principles*, 1996.

[2] D. Bobrow et al. TENEX, a paged time sharing system for the PDP-10. *Communications of the ACM*, 15(3), March 1972.

[3] J. Brustoloni and P. Steenkiste. Copy emulation in checksummed, multiple-packet communication. In *Proceedings of IEEE INFOCOM 1997*, pages 1124–1132, April 1997.

[4] H. Chartock and P. Snyder. Virtual swap space in SunOS. In *Proceedings of the Autumn 1991 European UNIX Users Group Conference*, September 1991.

[5] H. Chu. Zero-copy TCP in Solaris. In *Proceedings of the USENIX Conference*, pages 253–264. USENIX, 1996.

[6] C. Cranor. *Design and Implementation of the UVM Virtual Memory System*. Doctoral dissertation, Washington University, August 1998.

[7] P. Druschel and L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993.

[8] N. Ganapathy and C. Schimmel. General purpose operating system support for multiple page sizes. In *Proceedings of the USENIX Conference*. USENIX, 1998.

[9] R. Gingell, J. Moran, and W. Shannon. Virtual memory architecture in SunOS. In *Proceedings of USENIX Summer Conference*, pages 81–94. USENIX, June 1987.

[10] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 1997.

[11] M. Kaashoek et al. Application performance and flexibility on Exokernel systems. In *Proceedings of the Sixteenth Symposium on Operating System Principles*, October 1997.

[12] M. McKusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[13] J. Moran. SunOS virtual memory implementation. In *Proceedings of the Spring 1988 European UNIX Users Group Conference*, April 1988.

[14] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Operating Systems Design and Implementation (OSDI)*, pages 153–168, 1996.

[15] V. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. In *Operating Systems Design and Implementation (OSDI)*, pages 15–28. USENIX, 1999.

[16] The FreeBSD Project. The FreeBSD operating system. See `http://www.freebsd.org` for more information.

[17] The NetBSD Project. The NetBSD operating system. See `http://www.netbsd.org` for more information.

[18] R. Rashid et al. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computing*, 37(8), August 1988.

[19] O. Spatscheck and L. Peterson. Defending against denial of service attacks in Scout. In *Operating Systems Design and Implementation (OSDI)*, pages 59–72. USENIX, 1999.

[20] R. Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.

[21] L. Torvalds et al. The Linux operating system. See `http://www.linux.org` for more information.

Lightweight Structured Text Processing

Robert C. Miller and Brad A. Myers
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{rcm,bam}@cs.cmu.edu
http://www.cs.cmu.edu/~rcm/lapis/

Abstract

Text is a popular storage and distribution format for information, partly due to generic text-processing tools like Unix *grep* and *sort*. Unfortunately, existing generic tools make assumptions about text format (e.g., each line is a record) that limit their applicability. Custom-built tools are one alternative, but they require substantial time investment and programming expertise. We describe a new approach, *lightweight structured text processing*, which overcomes these difficulties by enabling users to define text structure interactively and manipulate the structure with generic tools. Our prototype system, LAPIS, is a web browser that can highlight, filter, and sort text regions described by the user. LAPIS has several advantages over other systems: (1) the ability to define custom structure with a simple, intuitive pattern language; (2) interactive specification, showing pattern matches in context and letting users choose the most convenient combination of manual selection and pattern matching; and (3) external parsers for standard text formats. The pattern language in LAPIS, *text constraints*, describes text structure in high-level terms, with region relationships like *before*, *after*, *in*, and *contains*. We describe an implementation of text constraints using a novel, compact representation of region sets as collections of rectangles, or *region intervals*. We also illustrate some examples of applying LAPIS to web pages, text files, and source code.

1 Introduction

Structured text has always been a popular way to store, process, and distribute information. Traditional examples of structured text include source code, SGML or LaTeX documents, bibliographies, and email messages. With the advent of the World Wide Web, structured text (in the form of HTML) has become a dominant medium for online information.

The popularity of text is easy to explain. As an old, standard data format, ASCII text can be viewed and edited easily on any platform. Text can be cut and pasted into any application, printed by any printer, included in any email message, and indexed by any search engine. Unix in particular has a rich set of generic tools for operating on text files: *grep*, *sort*, *uniq*, *sed*, etc.

Unfortunately, the generic nature of existing text-processing tools is also a weakness, because generic tools can make only limited assumptions about the format of the text. Most Unix tools assume that a text file is divided into records separated by newlines (or some other delimiter character). But this assumption breaks down for most kinds of structured text, such as source code and HTML. Consider the following tasks, which are difficult for generic text-processing tools to handle:

1. Find functions that call exit() in a program.
2. Check spelling in program comments.
3. Extract a bibliography from a Web page.
4. Sort a file of postal addresses by ZIP code.

The traditional approach to these problems is to custom-build a tool for a particular text format. For example, tasks #1 and #2 might be solved by a development environment customized for the programming language. Tasks #3 and #4 are typically solved by hand-coded Perl or AWK scripts. The problem with this approach is that custom-built programs require substantial investment, are difficult to reuse for other tasks or text formats, and lie beyond the ability of casual users to create.

The deficiencies of the custom-built approach are best highlighted by *custom* text structure – structure which has not been blessed by standard grammars or widely-available parsers. Many users store small databases (such as address lists) as text files. Many programs generate reports and logs in text form. Nearly every web page uses some kind of custom structure represented in HTML; examples include lists of publications, search engine results, product catalogs, news briefs, weather reports, stock quotes, sports scores, etc. Given the proliferation of custom text formats, developing a tool for every combination of task and text format is inconceivable.

Our approach to generic tools for structured text is called *lightweight structured text processing.* Lightweight structured text processing enables users to define custom text structure interactively and incrementally, so that generic tools can operate on the text in structured fashion. We envision that a lightweight structured text processing system would have four components:

- a *structure description language* for describing text structure;
- an *interactive document viewer* for viewing documents, developing and testing structure descriptions, and invoking tools;
- *parsers* for standard structures, like HTML and programming language syntax;
- *tools* for manipulating text using structure descriptions: sorting, searching, extracting, reformatting, editing, computing statistics, graphing, etc.

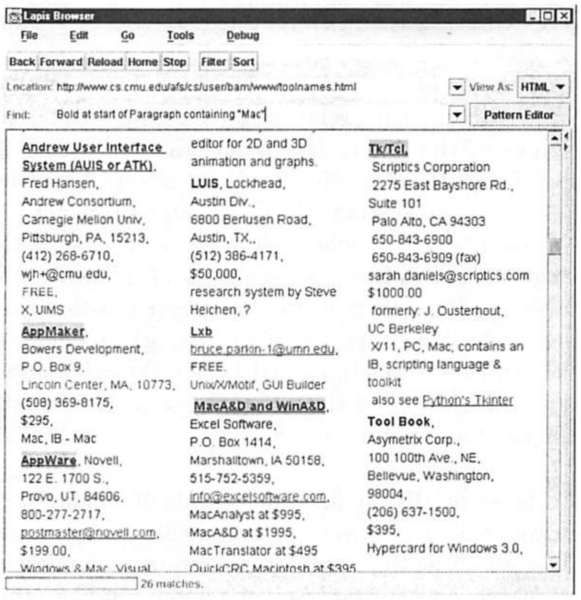
Following this plan, we have built a prototype system called LAPIS (Lightweight Architecture for Processing Information Structure). LAPIS includes a new structure description language called *text constraints.* Text constraints describe a set of regions in a document in terms of relational operators (like *before, after, in,* and *contains*) and primitive regions generated by external parsers or literal matching. Text constraints can be used not only for queries (such as `Function contains "exit"`) but also for structure definition, as in the following example:

```
Sentence = ends with SentencePunct;
SentencePunct = ('.' | '?' | '!'),
                just before Whitespace,
                but not '.' at end of
                      Abbreviation;
Abbreviation = 'Mr.' | 'Mrs.' |
               'Ms.' | 'Dr.' | ...;
```

Text constraints differ in several ways from context-free grammars and regular expressions (the traditional techniques for structure description). Text constraints permit conjunctions of patterns (indicated by commas in the previous example) and references to context (such as "just before"). Text constraints can also refer to structure defined by external parsers – even *multiple* parsers simultaneously. For example, `Line at start of Function` refers to both `Line` (a name defined by a line-scanning parser) and `Function` (defined by a programming-language parser) to match the first line of every function. Finally, we believe that text constraints are more readable and comprehensible for users than grammars or regular expressions, because a structure description can be reduced to a list of simple, intuitive constraints which can be read and understood individually. In the LAPIS prototype, text constraints are implemented as an algebra operating on sets of regions, using efficient set representations to achieve reasonable performance.

LAPIS combines text constraints with a web browser that allows the user to develop text constraints interactively and apply them to web pages, source code, and text files. In the browser, the user can describe a set of regions either programmatically (using text constraints or an external parser), manually (by selection), or using any combination of the two. Combining manual selection and programmatic description can be quite powerful. Manual selection can be used to restrict attention to part of a document which can be selected more easily than it can be described, such as the content area of a web page (omitting navigation bars and advertisements). Manual selection can also fix up errors made by an almost-correct structure description, adding or removing regions from the set as necessary. Relying on manual intervention is not always appropriate, but sometimes it can help finish a task faster.

The LAPIS browser also includes a few commands that operate on sets of regions. *Find* simply highlights and navigates through a set of regions. *Filter* displays only the selected regions, eliminating other text from the display. *Sort* displays a set of regions



Location: http://www.cs.cmu.edu/afs/cs/user/bam/www/toolnames.html

Find: Bold at start of Paragraph containing "Mac"

Andrew User Interface System (AUIS or ATK), Fred Hansen, Andrew Consortium, Carnegie Mellon Univ, Pittsburgh, PA, 15213, (412) 268-6710, wjh+@cmu.edu, FREE, X, UIMS

AppMaker, Bowers Development, P.O. Box 9, Lincoln Center, MA, 10773, (508) 369-8175, $295, Mac, IB - Mac

AppWare, Novell, 122 E. 1700 S., Provo, UT, 84606, 800-277-2717, postmaster@novell.com, $199.00, Windows & Mac, Visual

editor for 2D and 3D animation and graphs.

LUIS, Lockhead, Austin Div., 6800 Berlusen Road, Austin, TX,, (512) 386-4171, $50,000, research system by Steve Heichen, ?

Lxb bruce.parkin-1@umn.edu, FREE, Unix/X/Motif, GUI Builder

MacA&D and WinA&D, Excel Software, P.O. Box 1414, Marshalltown, IA 50158, 515-752-5359, info@excelsoftware.com, MacAnalyst at $995, MacA&D at $1995, MacTranslator at $495 QuickCRC Macintosh at $395

Tk/Tcl, Scriptics Corporation 2275 East Bayshore Rd., Suite 101 Palo Alto, CA 94303 650-843-6900 650-843-6909 (fax) sarah.daniels@scriptics.com $1000.00 formerly: J. Ousterhout, UC Berkeley X/11, PC, Mac, contains an IB, scripting language & toolkit also see Python's Tkinter

Tool Book, Asymetrix Corp., 100 100th Ave., NE, Bellevue, Washington, 98004, (206) 637-1500, $395, Hypercard for Windows 3.0,

26 matches.

Figure 1: The LAPIS web browser, showing a web page that describes user interface toolkits. The user has entered the pattern `Bold at start of Paragraph containing "Mac"` to highlight the names of toolkits that support Macintosh development.

sorted by the value of a subfield. In LAPIS, these features are provided as interactive commands in the browser, but we also plan to implement batch-mode tools in the style of *grep* and *sort*, which would take as input a text file and its structure description.

The remainder of this paper is structured as follows: Section 2 describes the LAPIS browser and tools. Section 3 describes the text constraints language. Section 4 describes our current implementation of text constraints. Section 5 presents some applications of the system to web pages, text files, and source code. Section 6 covers related work, Section 7 describes future work, and Section 8 concludes.

2 LAPIS Web Browser

Our prototype lightweight structured text processing system is LAPIS, a web browser that has been extended with a pattern language (text constraints) and several generic text-processing tools. LAPIS is built on top of Sun's Java Foundation Classes. A screenshot of the browser is shown in Figure 1.

Like other web browsers, the LAPIS browser can retrieve any file that can be named by a URL and retrieved by HTTP, FTP, or from the local filesystem. The browser can display text files or HTML pages. HTML pages can be displayed either as text, which shows the source including tags, or as HTML, which renders the page according to the HTML formatting.

Several parsers are included in the browser, which run automatically when a page of a certain MIME type is loaded. A *parser* interprets a particular text format and labels its components in the document. The built-in parsers include:

- **HTML:** parses HTML pages, labeling HTML tags and elements while simultaneously building a parse tree for rendering the page;
- **Character:** parses plain text and HTML to find character classes like `Whitespace`, `Letters`, and `Digits`;
- **Java:** parses Java programs to find syntax constructs like `Class`, `Method`, `Statement`, and `Expression`;
- **USEnglish:** parses plain text and HTML to find regions like `Sentence`, `Line`, `Time`, `Date`, and `Currency`, according to conventions of American English.

Parsers can also be associated with URL patterns. For example, a parser that identifies components of an AltaVista search result page might be associated with URLs of the form `http://altavista.digital.com/*`.

New parsers can be defined in two ways: writing a Java class that implements our `Parser` interface, or by developing a system of text constraints. The HTML and Character parsers were written by hand in Java. The Java parser was automatically generated from an example grammar included with the JavaCC parser-generator [26], showing that LAPIS can take advantage of existing parsers without recoding the grammar in text constraint expressions. USEnglish was developed interactively in LAPIS as a system of text constraints.

In the browser, the user can enter a text constraint expression and see the matching regions highlighted (see Figure 1). Highlighting is simple to implement and familiar to users, but unfortunately it merges adjacent and overlapping regions together, without distinguishing their endpoints. Future research should identify better ways to display overlapping region sets in context. To view highlighted regions, the user can either scroll the document or use the *Next Match* menu command to jump from one highlighted region to the next.

In addition to patterns, the user can also highlight regions by manual selection. In the prototype, a *selection* made with the mouse is distinct from the *highlighted region set* showing matches to a pattern. The selection is a single, contiguous region (colored blue), whereas the highlighted region set may be multiple, noncontiguous regions (colored red). The current selection in the document is always available as a one-element region set named `Selection`. By referring to `Selection` in a text constraint, for example, the user can limit the pattern's scope to a manually selected region of the document. The user can also construct a named region set by adding or removing regions. The *Label* menu command adds the current selection to the region set with the given name. A corresponding *Unlabel* command removes the selection from a given named region set by deleting regions that lie inside the selection and trimming the ends of regions that overlap the selection. By applying *Label* and *Unlabel* repeatedly to a sequence of selections, the user can build up a named region set by hand, or modify a named region set created by a parser or a pattern.

Several tools are provided for manipulating the highlighted regions. *Filter* eliminates all unhighlighted text from the display. By default, Filter inserts linebreaks between the highlighted regions to keep the display readable. Documents are filtered at the source text level - even HTML documents. The result is sometimes illegal HTML (with orphaned start tags or end tags), but the web browser can render it passably.

Like Filter, *Sort* filters the display down to highlighted regions, and also reorders the regions. Regions can be sorted alphabetically or numerically. By default, the sort key is the entire content of a region, but the user can provide an additional text constraint expression describing the sort field.

3 Text Constraints

Text constraints (TC) is a language for specifying text structure using relationships among regions (substrings of the text). TC describes a substring by specifying its start offset and end offset. Formally, a *region* is an interval $[b, e]$ of inter-character positions in a string, where $0 \leq b \leq e \leq n$ (n is the length of the string). A region $[b, e]$ identifies the substring that starts at the bth cursor position (just before the bth character of the string) and ends at the eth cursor position (just before the eth character, or at the end of the string if $e = n$). Thus the length of a region is $e - b$.

TC is essentially an algebra over sets of regions - operators take region sets as arguments and generate a region set as the result. TC permits an expression to match an arbitrary set of regions, unlike other structured text query languages that constrain region sets to certain types: nonoverlapping (regular expressions), nonnesting (GC-lists [5]), or hierarchical (Proximal Nodes [19]).

3.1 Primitives

TC has three primitive expressions: literals, regular expressions, and identifiers. A literal string enclosed in single or double quotes matches all occurrences of the string in the document. Thus `"Gettysburg"` finds all regions exactly matching the literal characters "Gettysburg". The literal matcher can generate overlapping regions, so matching `"aa"` against the string "aaaaa" would yield 4 regions.

A regular expression is indicated by `/regexp/`. Our regular expression matcher is based on the ORO-Matcher library for Java [20]. The library follows Perl 5 syntax and semantics [27], returning a set of nonoverlapping regions that are as long as possible.

An identifier is any whitespace-delimited token (except for words and punctuation reserved by TC operators). Identifiers refer to the named region sets generated by parsers. For example, after the HTML parser has run, `Tag` refers to the set of all HTML tags in the document. Only a single namespace is provided by the LAPIS prototype, so the names generated by different parsers must be chosen uniquely. A future version of LAPIS is expected to support multiple independent namespaces.

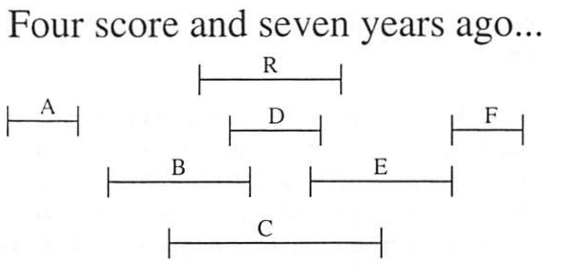


Figure 2: Fundamental region relations in an example string. Regions A through F are related to region R as follows: A *before* R; B *overlaps-start* R; C *contains* R; D *in* R; E *overlaps-end* R; and F *after* R.

3.2 Region Relations

TC operators are based on six fundamental binary relations among regions: *before*, *after*, *in*, *contains*, *overlaps-start*, and, *overlaps-end*. (Similar relations on time intervals were defined in [2].) The region relations are defined as follows:

$$\begin{array}{ll}
[b_1, e_1] \textit{ before } [b_2, e_2] & \Leftrightarrow e_1 \leq b_2 \\
[b_1, e_1] \textit{ after } [b_2, e_2] & \Leftrightarrow e_2 \leq b_1 \\
[b_1, e_1] \textit{ in } [b_2, e_2] & \Leftrightarrow b_2 \leq b_1 \wedge e_1 \leq e_2 \\
[b_1, e_1] \textit{ contains } [b_2, e_2] & \Leftrightarrow b_1 \leq b_2 \wedge e_2 \leq e_1 \\
[b_1, e_1] \textit{ overlaps-start } [b_2, e_2] & \Leftrightarrow b_1 \leq b_2 \wedge e_1 \leq e_2 \\
[b_1, e_1] \textit{ overlaps-end } [b_2, e_2] & \Leftrightarrow b_2 \leq b_1 \wedge e_2 \leq e_1
\end{array}$$

Note that *before* and *after* are inverses, as are *in* and *contains*, and *overlaps-start* and *overlaps-end*. The six region relations are illustrated in Figure 2.

The six region relations are complete in the sense that every ordered pair of regions is found in at least one of the relations. Some regions may be related in several ways, however. For example, in Figure 2, if A's end point were identical to R's start point, then we would have both A *before* R and A *overlaps-start* R. These relations are useful in pattern matching, so we define a set of derived relations in which regions have coincident endpoints:

$$\begin{array}{ll}
\textit{just-before} & = \textit{before} \cap \textit{overlaps-start} \\
\textit{just-after} & = \textit{after} \cap \textit{overlaps-end} \\
\textit{at-start-of} & = \textit{in} \cap \textit{overlaps-start} \\
\textit{at-end-of} & = \textit{in} \cap \textit{overlaps-end} \\
\textit{starts-with} & = \textit{contains} \cap \textit{overlaps-start} \\
\textit{ends-with} & = \textit{contains} \cap \textit{overlaps-end}
\end{array}$$

Figure 3 illustrates the derived relations.

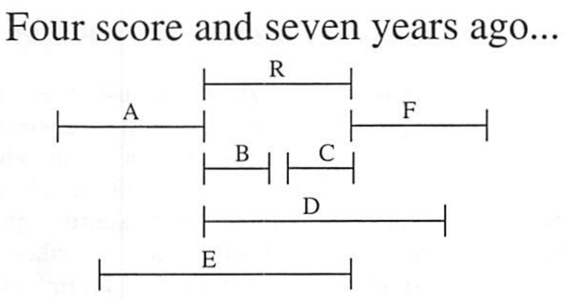


Figure 3: Region relations with coincident endpoints. Regions A through F are related to region R as follows: A *just-before* R; B *at-start-of* R; C *at-end-of* R; D *starts-with* R; E *ends-with* R; and F *just-after* R.

Another useful derived relation is `overlaps`:

$$\begin{array}{lll}
\textit{overlaps} & = & \textit{in} \cup \textit{contains} \cup \\
 & & \textit{overlaps-start} \cup \textit{overlaps-end}
\end{array}$$

In Figure 2, the regions B, C, D, and E *overlap* R, but A and F do not. In Figure 3, all the regions *overlap* R.

3.3 Relational Operators

Each region relation corresponds to a relational operator in TC. Each relational operator takes two forms, one *unary* and the other *binary*. The unary form, ***op*** S, generates the set of regions that bear the relation *op* to some region matching S. For example, in an HTML document, the constraint expression `in Paragraph` returns all regions that are inside some paragraph element.

The binary form of a relational operator, R ***op*** S, generates all regions matching R that bear the relation *op* to some region matching S. For example, in HTML, `Paragraph contains "Lincoln"` returns all paragraph elements that contain the string "Lincoln."

For the sake of simplicity, all relational operators have equal precedence and right associativity, so that `X in Y in Z` is parsed as `X in (Y in Z)`.

3.4 Intersection, Union, and Difference

Constraints that must be simultaneously true of a region are expressed by separating the constraint expressions with commas. The region set matched by S_1`,` S_2`,` `...`, S_n is the intersection of the region sets matched by each S_i. For example `just after "From:", just before "\n"` describes all regions that start immediately after a "From:" caption and end at a newline.

Alternative constraints are specified by separating the constraint expressions with "|". The region set matched by S_1 | S_2 | `...` | S_n is the union of the region sets matched by each S_i.

Set difference is indicated by `but not`. The region set matched by S_1 `but not` S_2 is the set that matches S_1 after removing all regions that match S_2.

3.5 Delimiter Operators

When certain relational operators are intersected, the resulting region set can be larger than the user anticipates. For example, the expression `starts with` R`, ends with` S matches every possible pair of R and S, even if other R's and S's occur in between. For situations where only adjacent pairs are desired, any relational operator can be modified by the keyword `delimiter`. For example, `starts with delimiter` S matches regions that start with some region matching S and overlap no other region matching S.

3.6 Concatenation and Background

Concatenation of regions is indicated by `then`. The expression `"Gettysburg" then "Address"` matches regions that consist of "Gettysburg" followed by "Address", with nothing important in between. The meaning of *nothing important* depends on a parameter called the *background*. The background is a set of regions. Characters in the background regions are ignored when concatenating constraint expressions. For example, when the background is `Whitespace`, the expression `"Gettysburg" then "Address"` finds not only "GettysburgAddress", but also "Gettysburg Address", and even "Gettysburg Address" split across two lines. Relational operators that require adjacency also use the background, so the expression `"Gettysburg" just before "Address"` will successfully match the first word of "Gettysburg Address".

The LAPIS browser chooses a default background based on the current document view, following the guideline that any text not printed on the screen is part of the background. In the plain text view, the default background is `Whitespace`. In the HTML view, the default background is the union of `Whitespace` and `Tag`, since tags affect rendering but are not actually displayed.

The background can also be set explicitly using the `ignoring` directive. To change the background to R for the duration of a constraint expression *expr*, use the form *expr* `ignoring` R. For example, a query on source code might take the form *expr* `ignoring (Comment | Whitespace)`. The background can be removed by setting it to `nothing`, which generates the empty region set.

3.7 Definitions and Constraint Systems

A *constraint definition* assigns a name to the result of a constraint expression:

```
GettysburgAddress =
    starts with
        "Four score and seven years ago",
    ends with
        "shall not perish from the earth"
```

Region sets named by a constraint definition can be used in the same way as region sets named by a parser, as in the example `Sentence at start of GettysburgAddress`. A *constraint system* is a set of constraint definitions separated by semicolons.

3.8 Expressiveness

The theoretical power of TC — that is, the set of languages that can be matched by a TC expression — depends on the power of the matchers and parsers it uses. If its matchers and parsers generate only regular languages, then the TC expression is also regular, since regular languages are closed under the TC operators concatenation, intersection, and union [11]. Since context-free languages are not closed under intersection, however, a TC expression using context-free parsers may match a non-context-free language.

A TC constraint system that uses only literals (no regular expressions or external parsers) is less powerful than a regular expression, because TC lacks recursive constraints or repetition operators (such as the $*$ operator). Future work discussed in Section 7 will address this issue.

4 Implementation

This section describes the implementation of text constraints used in LAPIS. Among the interesting features of the implementation is a novel region set representation, the *region interval*. Region intervals are particularly good at representing the result of a region relation operator. By a simple transformation, region intervals may be regarded as rectangles in two-dimensional space, allowing LAPIS to draw on earlier research in computational geometry to find a data structure suitable for storing and combining collections of region intervals.

4.1 Region Interval Representation

The key ingredient to an implementation of text constraints is choice of representation: how shall region sets be represented? One alternative is a bitvector, with one bit for each possible region in lexicographic order. With a bitvector representation, every region set requires $O(n^2)$ space, where n is the length of the document. Considering that the region sets generated by matchers and parsers typically have only $O(n)$ elements, the bitvector representation wastes space. Another alternative represents a region set as a list of explicit pairs $[b, e]$, which is more appropriate for sparse sets. Unfortunately the region sets generated by relational operators are *not* sparse. To choose a pathological example, `after` $[0, 0]$ matches every region in the document. In general, for any region relation `op` and region set S, the set matching `op` S may have $O(n^2)$ elements.

Other systems have dealt with this problem by restricting region sets to nested sets [19] or overlapped sets [5], sacrificing expressiveness for linear storage and processing. Instead of restricting region sets, we compress dense region sets with a representation called *region intervals*. A region interval is a quadruple $[b, c; d, e]$, representing the set of all regions $[x, y]$ such that $b \leq x \leq c$ and $d \leq y \leq e$. Essentially, a region interval is a set of regions whose starts and ends are given by intervals, rather than points. A region interval is depicted by extending the region notation for regions (|—|), replacing the vertical lines denoting the region's endpoints with boxes denoting intervals.

A few facts about region intervals follow immediately from the definition:

- The set of all regions in a string of length n can be represented by the region interval $[0, n; 0, n]$.
- The singleton region set $\{[b, e]\}$ is represented by the region interval $[b, b; e, e]$.
- A region interval represents the empty set if $b > c$ or $d > e$ or $b > e$.
- A region interval $[b_1, c_1; d_1, e_1]$ is a subset of another region interval $[b_2, c_2; d_2, e_2]$ if and only if $b_2 \leq b_1 \leq c_1 \leq c_2$ and $d_2 \leq d_1 \leq e_1 \leq e_2$.
- The intersection of two intervals $[b_1, c_1; d_1, e_1]$ and $[b_2, c_2; d_2, e_2]$ is

  $$[\max(b_1, b_2), \min(c_1, c_2); \max(d_1, d_2), \min(e_1, e_2)]$$

  which may of course be the empty set.

Region intervals are particularly useful for representing the result of applying a region relation operator. Given any region X and a region relation *op*, the set of regions which are related to X by *op* can be represented by exactly one region interval, as shown in Figure 4.

By extension, if a region relation operator is applied to a region set with m elements, then the result can be represented with m region intervals (possibly fewer, since some of the region intervals may be redundant).

This result extends to region intervals as well: applying a region relation operator to a region interval yields exactly one region interval. For example, the result of *before* $[b, c; d, e]$ is the set of all regions which lie before some region in $[b, c; d, e]$. Assuming the region interval is nonempty, every region ending at or before c qualifies, so the result of this operator can be described by the region interval $[0, c; 0, c]$.

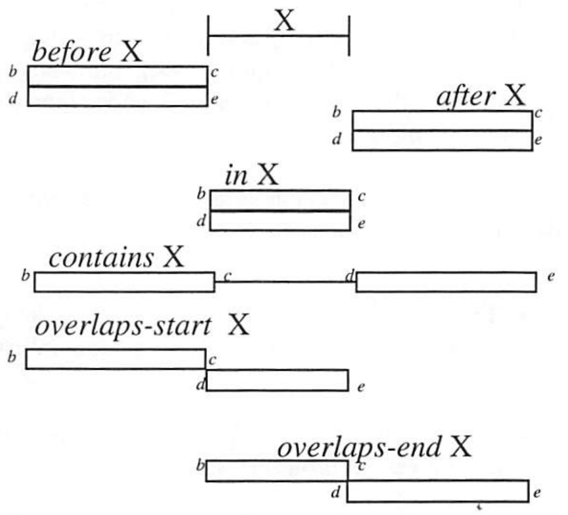


Figure 4: Region intervals corresponding to the relational operators.

We can represent an arbitrary region set by a union of region intervals, which is simply a collection of tuples. The size of this collection may still be $O(n^2)$ in the worst case (consider, for example, the set of all regions $[b, e]$ of even length, which must be represented by $O(n^2)$ singleton region intervals), but most collections are $O(n)$ in practice.

To summarize, the union-of-region-intervals representation enables a straightforward implementation of the text constraints language:

- **Primitives:** convert the set of regions generated by a literal, regular expression, or parser into a union of region intervals by replacing each region $[b, e]$ with the singleton region interval $[b, b; e, e]$.
- **Relational operators:** for each region interval $[b, c; d, e]$, compute a new region interval *op* $[b, c; d, e]$.
- **Union:** merge the two collections of region intervals, eliminating any region interval that is a subset of another.
- **Intersection:** intersect every possible pair of region intervals (one from each collection) and collect the results.

4.2 Region Space

It remains to choose a representation for the collection of region intervals that provides the operations we need (region relations, union, and intersection).

A 2D geometric interpretation of regions will prove helpful. Any region $[b, e]$ can be regarded as a point in the plane, where the x-coordinate indicates the start of the region and the y-coordinate indicates the end. We refer to this two-dimensional interpretation of regions as *region space* (see Figure 5). Strictly speaking, only points with integral coordinates correspond to regions, and even then only if they lie above the 45-degree line, where $b \leq e$.

Under this interpretation, a region interval $[b, c; d, e]$ corresponds to an axis-aligned rectangle in region space. Two region intervals intersect if and only if their region space rectangles intersect. A region interval is a subset of another if and only if its rectangle is completely enclosed in the other's rectangle.

A region set can be represented as a union of region intervals, which in turn can be represented as a union of axis-aligned rectangles in region space. We seek a data structure representing a union of rectangles with the following operations:

- **Create (P):** create a union of rectangles from a set of rectangles P.
- **Relation (op, R):** generate a new union of rectangles by applying a region relation operator *op* elementwise to R.
- **Union (R,S):** combine two unions of rectangles R and S.
- **Intersect (R,S):** intersect two unions of rectangles R and S.

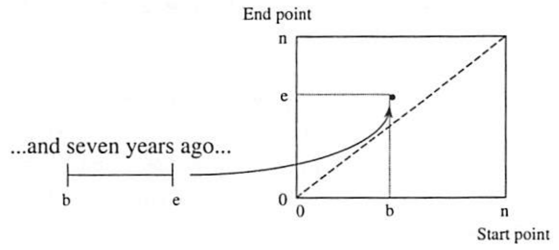


Figure 5: A region $[b, e]$ corresponds to a point in region space.

Ideally, these operations should take linear time and linear space. In other words, finding the intersection or union of a collection of M rectangles with a collection of N rectangles should take $O(N+M+F)$ time (where F is the number of rectangles in the result), and computing a region relation on a collection of N rectangles should take $O(N)$ time. The data structure itself should store N rectangles in $O(N)$ space.

Research in computational geometry and multidimensional databases has developed a variety of data structures and algorithms for storing and intersecting collections of rectangles, including plane-sweep algorithms, k-d trees, quadtrees of various kinds, R-trees , and R+-trees (see [24] for a survey).

LAPIS uses a variant of the R-tree [9]. The R-tree is a balanced tree derived from the B-tree, in which each internal node has between m and M children for some constants m and M. The tree is kept in balance by splitting overflowing nodes and merging underflowing nodes. Rectangles are associated with the leaf nodes, and each internal node stores the bounding box of all the rectangles in its subtree. The decomposition of space provided by an R-tree is adaptive (dependent on the rectangles stored) and overlapping (nodes in the tree may represent overlapping regions). To keep lookups fast, the R-tree insertion algorithm attempts to minimize the overlap and total area of nodes using various heuristics (for example, inserting a new rectangle in the subtree that would increase its overlap with its siblings by the least amount). One set of heuristics, called the R*-tree [4], has been empirically validated as reasonably efficient for random collections of rectangles. Initially we used the R*-tree heuristics in our prototype. The rectangle collections generated by text constraints are *not* particularly random, however; they tend to be distributed linearly along some dimension of region space, such as the 45-degree line, the x-axis, or the y-axis. We were able to improve overall performance by a factor of 5 by simply ordering the rectangles in lexicographic order, eliminating the expensive calculations that decide where to place a rectangle without sacrificing the tree's logarithmic decomposition of region space.

Two R-trees T_1 and T_2 can be intersected by traversing the trees in tandem, comparing the current T_1 node with the current T_2 node and expanding the nodes only if their bounding boxes overlap. Traversing the trees in tandem has the potential for pruning much of the search, since if two nodes high in each tree are found to be disjoint, the rectangles stored in their subtrees will never be compared. In practice, tandem tree intersection takes time $O(N+M+F)$. It will never do worse than $O(NM)$. Tandem tree traversal is effective for implementing set intersection, union, and difference.

4.3 Performance

The LAPIS prototype is written in Java 1.1. The core text constraints engine is implemented in about 3500 lines of code, not including matchers and parsers. The web browser consists of about 1000 lines of code on top of the JFC `JEditorPane` text component.

The text constraints engine can evaluate an operator at a typical rate of 20,000 regions per second, using Symantec JIT 3.0 on a 133 MHz Pentium. The actual evaluation time of a text constraint expression varies according to the complexity of the expression and the size of its intermediate results. The text constraint expressions used in the examples in Section 5 were all evaluated in less than 0.1 second, on text files and web pages ranging up to 80KB in size.

5 Applications

5.1 Web Pages

Many web pages display data in a custom format, using HTML markup to set off important parts of the text typographically or spatially. Figure 6 shows part of a page describing user interface toolkits [17]

The page describes over 100 toolkits with various properties: some are free, some are commercial; some run on Unix, others Microsoft Windows, others Macintosh, and others are cross-platform. To browse the page conveniently, we might want to restrict the display to show only toolkits matching certain requirements – for example, toolkits running under both Unix and Microsoft Windows, sorted by price.

Each toolkit on this page is contained in a single paragraph (<P> element in HTML). So we might start by describing the toolkit as the Paragraph element, which is identified by the built-in HTML parser:

```
Toolkit = Paragraph
```

AlphaWindow,
Cumulus Technology Corp.,
1007 Elwell Court,
Palo Alto, CA, 94303,
(415) 960-1200,
$750,
Unix, **Discontinued**,
Alpha-numeric terminal windows, Window System

Altia Design, Altia,
5030 Corporate Plaza Dr #300,
Colorado Springs, CO, 80919,
(800)653-9957 or (719)598-4299,
UNIX or Windows, IB

Amulet,
Brad Myers,
Human-Computer Interaction Institute,
Carnegie Mellon Univ,
Pittsburgh, PA, 15213,
(412) 268-5150,
amulet@cs.cmu.edu,
FREE,
X or MS Windows, portable toolkit, UIMS

Figure 6: Excerpt from a web page describing user interface toolkits.

Finding the prices is straightforward using `Number`, a region set identified by the built-in `USEnglish` parser:

```
Price = ("\$" then Number | "FREE")
        in Toolkit;
```

Finding toolkits that run under Macintosh is easy (`Toolkit contains "Mac"`), since the page refers consistently to Macintosh as "Mac". But Unix platforms are sometimes described as "X", "X Windows", or "Motif", and Microsoft Windows is also called "MS Windows" or just plain "Windows". We deal with these problems by defining a constraint for each kind of platform that specifies all these possibilities and further constrains the matched literal to be a full `Word` (not just part of a word):

```
Macintosh = Word, "Mac";
Unix = Word, ("Unix" | "X" | "Motif");
MSWindows = Word, ("PC" |
      "Windows" but not just after "X");
```

Using these definitions, we can readily filter the web page for toolkits matching a certain requirements (`Toolkit, contains Unix, contains MSWindows`) and sort them according to `Price`.

5.2 Plain Text

Plain text has less explicit structure than HTML, so text constraints for plain text typically refer to delimiters like punctuation marks and line breaks. Consider the following example of processing email messages. Several airlines distribute weekly email announcing low-price airfares. An excerpt from one message (from US Airways) is shown in Figure 7.

Describing the boundaries of the table itself is fairly straightforward given the delimiters (`BlankLine` is identified by the built-in `USEnglish` parser):

```
Table = starts with delimiter
            "Roundtrip Fares Departing From",
        ends with delimiter BlankLine;
```

The rows of the table can be found using `Line`, also identified by the built-in parser:

```
Flight = Line starts with "\$" in Table;
Fare = Number just after "\$" in Flight;
```

The origin and destination cities can be described in terms of their boundaries:

```
Origin = just after delimiter "From",
         just before delimiter "To",
         in Line at start of Table;
Destination = just after Price,
          in Flight;
```

```
Roundtrip Fares Departing From BOSTON, MA To
-------------------------------------------------
        $109                INDIANAPOLIS, IN
        $89                 PITTSBURGH, PA

Roundtrip Fares Departing From PHILADELPHIA, PA To
-------------------------------------------------
        $79                 BUFFALO, NY
        $89                 CLEVELAND, OH
        $89                 COLUMBUS, OH
        $89                 DAYTON, OH
        $89                 DETROIT, MI
        $79                 PITTSBURGH, PA
        $79                 RICHMOND/WMBG., VA
        $79                 SYRACUSE, NY
```

Figure 7: Excerpt from an email message announcing cheap airfares.

```
/**
 * Convert a local filename to a URL.
 * @param file File to convert
 * @return URL corresponding to file
 */
public static URL FileToURL (File file)
        throws MalformedURLException {
    return new URL ("file:"
        + toURLDelimiters
            (file.getAbsolutePath ()));
}
```

Figure 8: A Java method with a documentation comment.

Using these definitions, we can readily filter the message for flights of interest, e.g. from Boston to Pittsburgh:

```
Flight,
contains Destination contains "PITTSBURGH",
in Table contains Origin contains "BOSTON";
```

The expression for the flight's origin is somewhat convoluted because flights (which are rows of the table) do not contain the origin as a field, but rather inherit it from the heading of the table. This example demonstrates, however, that useful structure can be described and queried with a small set of relational operators.

5.3 Source Code

Source code can be processed like plain text, but with a parser for the programming language, source code can be queried much more easily. LAPIS includes a Java parser, so the examples that follow are in Java.

Unlike other systems for querying and processing source code, TC operates on regions in the source text, not on an abstract syntax tree. At the text level, the user can achieve substantial mileage knowing only a few general types of regions identified by the parser, such as `Statement`, `Comment`, `Expression`, and `Method`, and using text constraints to specialize them. For example, our parser identifies `Comment` regions, but does not specially distinguish the "documentation comments" that can be automatically extracted by the `javadoc` utility. Figure 8 shows a Java method preceded by a documentation comment.

The user can find the documentation comments by constraining `Comment` with a text-level expression:

```
DocComment = Comment starts with "/**";
```

A similar technique can be used to distinguish public class methods from private methods:

```
PublicMethod = Method starts with "public";
```

In this case, however, the accuracy of the pattern depends on programmer convention, since attributes like `public` may appear in any order in a method declaration, not necessarily first. All of the following method declarations are equivalent in Java:

```
public static synchronized void f ()
static public synchronized void f ()
synchronized static public void f ()
```

If necessary, the user can deal with this problem by adjusting the pattern (e.g., `Method starts with Line contains "public"`) or relying on the Java parser to identify attribute regions (e.g., `Method contains Attribute contains "public"`). In practice, however, it is often more convenient to use typographic conventions, like `public` always appearing first, than to modify the parser for every contingency. Since text constraints can express such conventions, constraints might also be used to enforce them, if desired.

We can use `DocComment` and `PublicMethod` to find public methods that need documentation:

```
PublicMethod but not just after DocComment;
```

Text constraints are also useful for defining custom structure inside source code. Java documentation comments can include various kinds of fields, such as `@param` to describe method parameters, `@return` to describe the return value, and `@exception` to describe exceptional return conditions. These fields can be described by text constraint expressions:

```
DocField = starts with delimiter "@",
           in DocComment;
ParamDoc = DocField, starts with "@param";
ReturnDoc = DocField, starts with "@return";
ExceptionDoc = DocField, starts with
                            "@exception";
```

Using this structure, we can find methods whose documentation is incomplete in various ways. For example, this expression finds methods with parameters but no parameter documentation:

```
PublicMethod contains FormalParameter,
  just after (DocComment but not
                 contains ParamDoc);
```

6 Related Work

Text processing is a rich and varied field. Languages like AWK [1] and Perl [27] are popular tools providing fast regular expression matching in an imperative programming language designed for text processing. These tools are not interactive, however, sacrificing the ability to view pattern matches in context (particularly important for web pages) and the ability to combine manual selection with programmatic selection. Visual Awk [15] made some strides toward interactive development of AWK programs which was inspirational for this work, but Visual AWK is still line-oriented, limited to regular expression patterns, and unable to use external parsers.

The concept of lightweight structured text processing described in this paper is independent of the language chosen for structure description. The text constraints language in LAPIS is novel and appealing for its simple and intuitive operators, its uniform treatment of parser-generated regions and constraint-generated regions, the concept of background regions, and its direct implementation, but another language may be used instead. A variety of languages have been proposed for querying structured text databases, such as Proximal Nodes [19], GC-lists [5], p-strings [8], tree inclusion [13], Maestro [16], and PAT expressions [23]. A survey of structured text query languages is found in [3]. Sgrep [12] is a variant of *grep* that uses a structured text query language instead of regular expressions, which helped inspire us to incorporate other Unix-style tools into a structured text processing system. Domain-specific query tools include ASTLOG [6], a query language specific to source code, and WebL [14], which combines an HTML query language with a programming language specialized for fetching and processing World Wide Web pages.

Structured text editors are a common form of structured text processing, but lacking the "lightweightness" that enables users to construct structure descriptions interactively. Examples of structured text editors include Gandalf [10], GRIF [22], and to some extent, EMACS [25]. These systems accept a structure description and provide tools for editing documents that follow the structure. The structure description is generally a variant of context-free grammar, although EMACS uses regular expressions to describe syntax coloring. EMACS is unusual in another sense, too: unlike structured text editors that enforce syntactic correctness at all times, EMACS uses the structure description to assist editing where possible, but does not prevent the user from entering free text. Our LAPIS system follows this philosophy, allowing the user to describe and access the document as free text, as structured text, or any combination of the two.

Sam [21] combines an interactive editor with a command language that manipulates regions matching regular expressions. Regular expressions can be pipelined to automatically process multiline structure in ways that line-oriented systems cannot. Unlike LAPIS, however, Sam does not provide mechanisms for naming, composing, and reusing the structure described by its regular expressions.

Also related are recent efforts to build structure-aware user interfaces, such as Cyberdesk [7] and Apple Data Detectors [18]. These systems associate actions with text structure, so that URLs might be associated with the "open in browser" action, and email addresses with "compose a message" or "look up phone number." When a URL or email address is selected by the user, its associated actions become available in the user interface. Action association is a useful tool that might be incorporated in LAPIS, but unlike LAPIS, these other systems use traditional structure description languages like context-free grammars and regular expressions.

7 Future Work

This work is part of the first author's PhD thesis research, and continues to evolve. This section describes some of the directions in which the work will be taken in the coming months.

LAPIS will be extended with new matchers, parsers, and tools. A more useful matcher for literals would optionally ignore alphabetic case, optionally match only full words, match spaces in the literal expression against any background character, and optionally do simple stemming. Parser support would be improved by allowing parsers to operate on limited parts of the document - for example, applying an HTML parser only to Java documentation comments, which may contain HTML tags. Useful new tools would include computing statistics on region sets (such as counts, sums, and averages) and reformatting text by template substitution.

Another fruitful area for research is integration of lightweight structured text processing into other applications, in particular an extensible text editor such as EMACS. Integration with a text editor poses at least two challenges: the interface problem of using named region sets fluidly in direct-manipulation text editing, and the implementation problem of updating region sets cheaply as the user edits.

The text constraint language has room for improvement. It should be possible to count (e.g. `2nd Line in Table`) and use numeric operators (e.g. `Toolkit contains Price < 100`). Constraint systems should support recursive or mutually recursive definitions. It would also be useful to precede a constraint expression by a *fuzzy qualifier*, such as `always`, `usually`, `rarely`, or `never`. A fuzzy qualifier describes how important it is for a matching region to satisfy the constraint. Finally, it will be important to determine the conditions under which our text contraints implementation (tandem tree intersection) runs in linear time.

8 Conclusions

This paper has described lightweight structured text processing, a technique for allowing users to define and manipulate text structure interactively. A prototype system, LAPIS, was described and evaluated on example applications, including web pages, source code, and plain text. LAPIS includes a structure description language called text constraints, which can express text structure in terms of relationships among regions.

The LAPIS prototype has several important advantages over other systems. First is the ability to handle *custom structure* with a simple language accessible to users. The second advantage is *interactive specification*, which allows users to see pattern matches in context and define text structure by the most convenient combination of manual selection and pattern matching. Finally, LAPIS supports *external parsers*, giving the user leverage over standard text formats, supporting existing parsers without recoding them in a new grammar language, and allowing the user to write patterns that refer to multiple parse trees at once.

Availability

The LAPIS prototype described in this paper, including Java source code, is available free from `http://www.cs.cmu.edu/~rcm/lapis/`.

Acknowledgements

For help with this paper, the authors would like to thank David Garlan, Laura Cassenti, and the anonymous referees.

This research was partially supported by a USENIX Student Research Grant, and partially by a National Defense Science and Engineering Graduate Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied of the U.S. Government.

References

[1] Aho, A.V., Kernighan, B.W., and Weinberger, P.J. *The AWK Programming Language.* Addison-Wesley, 1988.

[2] Allen, J. "Time Intervals." *Communications of the ACM*, v26 n11, 1983, pp 822-843.

[3] Baeza-Yates, R. and Navarro, G. "Integrating contents and structure in text retrieval." *ACM SIGMOD Record*, v25 n1, March 1996, pp 67-79.

[4] Beckmann, N., Kriegel, H-P., Schneider, R., and Seeger, B. "The R*-tree: an efficient and robust access method for points and rectangles." *ACM SIGMOD Intl Conf on Managment of Data*, 1990, pp 322-331.

[5] Clarke, C.L.A., Cormack, G.V., Burkowski, F.J. "An algebra for structured text search and a framework for its implementation." *The Computer Journal*, v38 n1, 1995, pp 43-56.

[6] Crew, R. F. "ASTLOG: a language for examining abstract syntax trees." *Proceedings of the*

USENIX Conference on Domain-Specific Languages, October 1997, pp 229-242.

[7] Dey, A.K., Abowd, G.A., and Wood, A. "CyberDesk: a framework for providing self-integrating ubiquitous software services." *Proceedings of Intelligent User Interfaces '98*, January 1998.

[8] Gonnet, G. H. and Tompa, F. W. "Mind your grammar: a new approach to modelling text." *Proceedings 13th VLDB Conference*, 1987, pp 339-345.

[9] Guttman, A. "R-Tree: a dynamic index structure for spatial searching." *ACM SIGMOD Intl Conf on Managment of Data*, 1984, pp 47-57.

[10] Habermann, N. and Notkin, D. "Gandalf: Software development environments." *IEEE Transactions on Software Engineering.* v12 n12, December 1986, pp 1117-1127.

[11] Hopcroft, J.E. and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[12] Jaakkola, J. and Kilpelainen, P. *Using sgrep for querying structured text files.* University of Helsinki, Department of Computer Science, Report C-1996-83, November 1996.

[13] Kilpelainen, P. and Mannila, H. "Retrieval from hierarchical texts by partial patterns." *Proceedings SIGIR '93*, pp 214-222, 1993.

[14] Kistler, T. and Marais, H. "WebL - a programming language for the Web." In *Computer Networks and ISDN Systems (Proceedings of the WWW7 Conference)*, v30, April 1998, pp 259-270. Also appeared as DEC SRC Technical Note 1997-029.

[15] Landauer, J. and Hirakawa, M. "Visual AWK: a model for text processing by demonstration." *Proceedings 11th International IEEE Symposium on Visual Languages '95*, September 1995. `http://www.computer.org/conferen/vl95 /talks/T32.html`

[16] MacLeod, I. "A query language for retrieving information from hierarchic text structures." *The Computer Journal,* v34 n3, 1991, pp 254-264.

[17] Myers, B.A. *User Interface Software Tools.* `http://www.cs.cmu.edu/~bam/toolnames.html`

[18] Nardi, B.A., Miller, J.R., and Wright, D.J. "Collaborative, programmable intelligent agents." *Communications of the ACM*, v41 n3, March 1998, pp 96-104.

[19] Navarro, G. and Baeza-Yates, R. "A language for queries on structure and contents of textual databases." *Proceedings SIGIR'95*, pp 93-101.

[20] Original Reusable Objects, Inc. *OROMatcher.* `http://www.oroinc.com/`

[21] Pike, R. "The Text Editor sam." *Software Practice & Experience*, v17 n11, Nov 1987, pp 813-845.

[22] Quint, V. and Vatton, I. "Grif: an interactive system for structured document manipulation." *Text Processing and Document Manipulation, Proceedings of the International Conference*, Cambridge University Press, 1986, pp 200-213.

[23] Salminen, A. and Tompa, F. W. *PAT expressions: an algebra for text search.* UW Centre for the New Oxford English Dictionary and Text Research Report OED-92-02, 1992.

[24] Samet, H. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[25] Stallman, R.M. "EMACS - the extensible, customizable self-documenting display editor." *SIGPLAN Notices*, v16 n6, June 1981, pp 147-56.

[26] Sun Microsystems, Inc. *JavaCC.* `http://www.suntest.com/JavaCC/`

[27] Wall, L., Christiansen, T., and Schwartz, R.L. *Programming Perl*, 2nd ed. O'Reilly & Associates, 1996.

SBOX: Put CGI Scripts in a Box

Lincoln D. Stein
Cold Spring Harbor Laboratory
Cold Spring Harbor, NY, 11724
lstein@cshl.org, http://stein.cshl.org

Abstract

sbox is a CGI wrapper script that allows Web sites to safely grant CGI authoring privileges to untrusted or naive authors. The script increases security in several ways. It changes the process privileges of CGI scripts to match their owners, preventing one script from interfering with another's data files or operations. It establishes configurable ceilings on script resource usage, avoiding intentional or unintentional denial of service attacks. Most importantly, *sbox* can also be used to run untrusted CGI scripts within a *chroot()*-ed directory, thereby preventing CGI scripts from accessing sensitive portions of the file system.

sbox can be used and redistributed freely. The complete package is available for download at

http://stein.cshl.org/WWW/software/sbox/

1 Introduction

Common Gateway Interface (CGI) scripts were among the first techniques for creating interactive Web pages and probably remain the most popular [Stein97]. Perhaps the main reason for the enduring popularity of CGI scripts is their simplicity. To create a dynamic Web page, a Web author writes a program that prints a short HTTP header followed by the contents of the desired Web page. The author then moves the program into a specially designated "CGI directory" on the Web server host. When the program's URL is requested, its output is displayed on the Web page.

A CGI script can be written in any language, compiled or interpreted. A fully functional CGI script can be written in just three lines of Bourne shell scripting code (including the #! line):

```
#!/bin/sh
echo -e "Content-type: text/plain\n"
echo "Hello world!"
```

The full CGI protocol [Coar] provides mechanisms for scripts to accept input from web forms, get information about the current operation of the server, learn the name and IP address of the remote browser, and pass back status information to the Web server. Communication between script and server is accomplished via environment variables and standard input/output. Essentially, the CGI protocol is a transient coshell system [Fowler] in which the Web server delegates the responsibility of producing the page content onto an external program, the script.

1.1 Security Problems with CGI Scripts

The simplicity and ease with which CGI scripts can be created is also the protocol's Achille's heel [Garfinkle, Rubin, Stein98a]. It is so simple to write CGI scripts that programming novices who have no prior experience in network server software development can readily create interactive Web pages. And here's where the problem lies. Novices, and sometimes even experienced programmers, are prone to errors that expose the Web server host to attack by unscrupulous individuals.

As an example of the problems beginners run into, consider the following CGI script written in Perl. Its intent is to recover an e-mail address from a submitted fill-out form and then mail a message to that address using the *mail* program.

```
#!/usr/local/bin/perl
use CGI qw(:standard);

$mailto   = param('mailto');
$subject  = param('subject');
$contents = param('contents');

open (MAIL,"|mail -s $subject $mailto");
print MAIL $contents";
print MAIL ".\n";
close MAIL;
print "Content-type: text/plain\n\n"
print "Mail sent to $address.";
```

This script, which is intended to be typical of a beginner's program rather than illustrative of good style, begins by using the *param()* function of the Perl CGI module [Stein98b] to recover the contents of three HTML form fields named "mailto," "subject," and "contents." The values of "mailto" and "subject" are used to open up a pipe to the Unix *mail* command. The value of "contents" is then printed to the *mail* process, which is then closed. The script ends by printing out a short confirmation message.

This script has a number of problems, including a reliance on the PATH environment variable to resolve the **mail** command, a failure to examine the "contents" field for a line beginning with a dot, which would terminate the mail message prematurely, and a failure to check for errors after the *open* and *close()* calls. However the most egregious flaw is in the call to *open()*, whre the programmer passes the contents of "subject" and "mail" to the shell without having first checked them for metacharacters. Consider what happens when the wiley hacker provides the following text as the value of the "mailto" field:

```
hackers_r_us@hackers.com</etc/passwd
```

The piped *open()* transforms this into the following call:

```
mail hackers_r_us@hackers.com</etc/passwd
```

with the result that the system password file is inadvertently mailed out to a potential attacker.

Other common problems in CGI scripts include the failure to check the length of strings before copying them into static buffers, failure to check for the existence of temporary files before clobbering them, and the failure to check user-provided pathnames for the ".." characters before opening files. Possible CGI script exploits include a variety of denial of service attacks. For example, a CGI script that reads user-provided input and spools it to a disk file is vulnerable to the mischievious hacker who uses a Web robot to transmit an endless stream of random bits to the script. Eventually the server's file system will fill up, causing the host system to stall.

Experience has shown that CGI scripts are a major source of vulnerability on the Web. Over the past five years, dozens well-known and widely distributed CGI scripts have been found to contain exploitable security holes [Stein98c, CERT]. Even experienced developers get burned from time to time. Offenders include freeware/public domain scripts, such as *count.cgi* as well as commercial products from such respected developers as Microsoft and Silicon Graphics. Understandably, most Webmasters are extremely cautious about installing new and untested CGI scripts on their servers.

One way to limit the harm that poorly-written CGI scripts can do is to run the Web server with as few privileges as possible. Most Web servers run as an unprivileged user without login privileges, such as "nobody." CGI script processes spawned by the server will ordinarily run under the same privileges as their parent, and by carefully controlling file and directory permissions, the Webmaster can limit the scope of any potential damage that errant CGI scripts can inflict on the server host. To increase safety even further, the Webmaster could place the entire server into a restricted directory using the **chroot** command. Now any CGI scripts it spawns will be limited to the portion of the file system that the server runs in.

1.2 User-Maintained CGI Scripts

Now consider a Web server run in an academic environment or by an Internet service provider (ISP). Such a system generally supports multiple Web authors of varying levels of experience and aptitude. In an academic environment, the authors are students, faculty members, and support staff who are granted personal Web pages. In the case of an ISP, the users are customers who have paid for Web space, and can range from individuals who maintain personal "vanity" pages to large co-hosted corporations. If users are allowed to write and install their own CGI scripts, then the risk from user-maintained scripts is magnified several fold.

First of all, a malicious author might seek to break into the Web server host by writing a CGI script that deliberately probes the host for holes. In many Web hosting environments, authors are not given a login shell. Instead they are constrained to uploading new and modified HTML pages via FTP or a Web publishing package such as FrontPage [Microsoft]. If authors are allowed to upload Perl scripts and compiled binaries for use as CGI scripts, this policy is easily circumvented.

Second, even if the host is protected by running the Web server as an unprivileged user and in a change-root directory, there is nothing to protect authors from each others' CGI scripts. Because all CGI scripts run under the same user account and execute in the same change-root directory, there is nothing protecting one author's data from another author's script. A student could write a CGI script to peek at the answers to a faculty member's online quiz, kill other students' CGI processes, or fill a user's guestbook file with obscene messages. In an ISP environment, one corporate customer could write a CGI script to spy on another customer's order entry system and client database.

Third, even if there is no active intent to do evil on the part of an author, a single poorly-written CGI script can still be used by Internet intruders to compromise the security of all authors on the system. For example, a guestbook script that doesn't check for the presence of ".." directories in the path to its data file can easily be exploited to view or overwrite files maintained by other authors.

Fourth, user-maintained CGI scripts are an invitation to denial of service (DoS) attacks. A malicious script writer can launch a DoS attack on the Web server host with a Perl script like the one shown below, which forks itself forever until the server host runs out of slots in its process table. It is possible that the system administrator will be unable to log in to kill the runaway process and may be forced to reboot the machine:

```
#!/usr/local/bin/perl
fork() while 1;
```

Finally, it is difficult to trace an attack from a user-maintained CGI script back to its owner. Since all scripts execute with the identity and privileges of the Web server, there is no easy way to determine whose script is, for example, leaving 40 megabyte scratch files in */tmp*.

1.3 Wrappers

There are a number of approaches to the problem of user-maintained CGI scripts. One approach is to outlaw them completely. The site's administrators can preinstall a number of standard CGI scripts for users to link to and configure the server so that no additional scripts can be added. Another solution is to submit all user-written scripts to an exacting code review.

Neither of these approaches is particularly appealing. The first solution is unlikely to be popular in the competitive Web hosting market where customers migrate to the service that offers the most features for the least cost. The second solution is only practical for sites that have unusually generous administrative resources or an unusually small number of users who want to install custom CGI scripts.

A more practical solution is to use a *wrapper* script. Instead of invoking user-maintained CGI scripts directly, the web server runs then indirectly via a wrapper program. The wrapper modifies the environment in some way that make the execution of the user-maintained script safer. The wrapper is also a good place to enforce security policy decisions. For example, the wrapper can keep a log of the scripts it has run and can refuse to run scripts whose permissions are insecure.

The first and still most widely-used wrapper program was *cgiwrap*, written by Nathan Neulinger [Neulinger]. *cgiwrap* performs several useful functions. Its main feature is that it uses the Unix *setuid()* call to run user-maintained CGI scripts under the user and group ID of the script's owner rather than the shared Web server account. This prevents one user's scripts from writing to data files maintained by another, and makes it easier to track down problems caused by poorly written scripts. *cgiwrap* also allows the Webmaster to place resource limitations on user-maintained scripts using the Berkeley *setrlimit()* call. This prevents a number of deliberate and inadvertent DoS attacks.

The *cgiwrap* program is straightforward to use. Once *cgiwrap* is installed in the system CGI directory, URLs used to invoke user-maintained scripts like this one:

http://www.site.com/~fred/guestbook.cgi

are replaced by URLs that invoke *cgiwrap*:

http://www.site.com/cgi-bin/cgiwrap/fred/guestbook.cgi

More recently, the popular Apache Web server has shipped with a built-in wrapper program called *suEXEC* [Apache Group]. The operation of *suEXEC* is similar to *cgiwrap*, but it is more tightly integrated into the Web server, making it unecessary to change any URLs in order to use it. In addition to changing its user ID to match that of the owner of the script, *suEXEC* logs each script it executes along with the user and group ID that it runs under. It also performs a series of consistency checks in order to detect unsafe practices. For example, *suEXEC* will refuse to run a script that is world writable or which is contained within a world writable directory.

The main limitation of both *cgiwrap* and *suEXEC* is that neither truly insulates scripts written by one user from those written by another. Naive users who store confidential information in world readable files and directories can still be attacked when another user's CGI script is used to peek at that data. In fact, although these scripts increase the security of the Web hosting service as a whole, they decrease the security of the individual user. Because the wrapped script runs with the same privileges as the user, it has free access to all the user's files. A poorly written script can be tricked into changing the user's HTML documents or recursively deleting his home directory. It can also impersonate the user, for example by sending e-mail from the user's account.

2 The *sbox* Wrapper

The *sbox* program is a CGI wrapper that goes beyond *cgiwrap* and *suEXEC* to offer the following features:

1. *sbox* calls *suid()* to run the requested script with the privileges of the owner of the script or the script's containing directory.
2. It calls *sgid()* to run the requested script under the privileges of the group that owns the script or the script's containing directory.
3. It performs consistency checks on the script file and directory ownerships to catch insecure situations such as world-writable scripts.
4. It establishes limits on the script's use of CPU, memory, processes, files and other resources.
5. It calls *chroot()* to run the target CGI script in a restricted change-root directory locatated within the user's home directory.
6. It cleanses the environment of information that is not germaine to CGI scripts.
7. It logs its actions and executes the target script.

These features can be used together, or can be switched on and off selectively to implement a variety of security policies.

Once installed, *sbox* is straightforward to use. To run an untrusted CGI script, create a composite URL consisting of the path to *sbox* followed by the path to the target CGI script. A typical URL for invoking a user-supported script looks like this:

http://www.site.com/cgi-bin/sbox/~fred/guestbook.cgi

sbox can also be used in conjunction with the virtual hosts feature provided by Apache and other servers. With some servers, it is even possible to make *sbox* transparent, so that its name doesn't appear in the path. A scheme to do this using the Apache *mod_rewrite* module is presented later in this paper.

The next sections describe each of *sbox*'s features in more detail and shows how they can be used to increase the security of the Web site.

2.1 suid()/sgid() Features

Before *sbox* launches a user-supported CGI script, it can be configured to change its UID and/or GID to match the script's owner. There are two possible variants of this feature. In the first variant, *sbox* uses the script file to determine which user and group to run as. This functionality is similar to the scheme implemented by *cgiwrap*. In the second variant, the ownership of the script is ignored; instead the ownership of the directory that contains it is used to determine the user and/or group.

Allowing *sbox* to take on the identity of the enclosing directory might seem a bit obscure, but the rationale is that it gives the Webmaster more flexibility than just using the script ownership does. For example, the Webmaster could use this technique to create a common *cgi-bin* directory for use by a particular group of developers. The directory would be owned by a pseudo-user and be group writable by each of the developers, allowing any user in the group to create and edit CGI scripts. When the script runs, it executes under the permissions of the common pseudo-user account, preventing it from modifying any of the author's files or databases unless he explicitly gives it permission to do so by setting the group writable bit.

Another strategy that the Webmaster might want to adopt is to configure *sbox* so that it performs an *sgid()* only. This will cause the target script to be executed with the group permissions of the script or enclosing directory, but with the user permissions of the Web server. By adopting a system-wide user-private group strategy in which each user is assigned a unique primary group, the script's author can exactly control what resources the script does and does not have access to. This strategy also makes it possible to create scripts that cannot modify their own source code file or binary, a risk both *cgiwrap* and *suEXEC* are subject to.

2.2 Consistency Checks

When *sbox* launches, it checks its environment for signs that it has been tampered with or that it is being run in an unsafe fashion. If any of the checks fail, *sbox* aborts with an error message.

The following checks are performed:

1. *sbox* checks that it was launched by the unprivileged user and group that the Web server runs as, for example **nobody** and **nogroup**. This check is to avoid the possibility that some user or group is trying to exploit the script's set-user-id features from the command line.
2. It checks whether it was launched by the root user, and aborts if so. This is often a sign that the Web server is misconfigured.
3. It checks the target CGI script for set-user-id and set-group-id bits and refuses to run if so. Untrusted users shouldn't be allowed to write suid or sgid scripts.
4. It checks that the target CGI script is executable by other, and aborts if not, assuming that the script's author had some reason for turning off the world execute bit.
5. It checks that neither the target CGI script nor its enclosing directory is owned by unprivileged user and group that the web server runs as. If the target is owned by this user, it's possible that it is a trojan horse created by a file upload script.
6. It checks that neither the target file nor its enclosing directory is world writable.

7. If the *chroot()* feature is active, it checks that the target script is located within the directory that will become the new root. This is a prerequisite for launching the script after the *chroot()* call.

8. Lastly, *sbox* checks that the target and its enclosing directory are owned by users and/or groups in an approved range, usually high-numbered IDs. This prevents *sbox* from being tricked into running a script as a special user such as **bin**.

These checks, along with the environment sanitization performed later in the launch process, go a long way toward preventing many of the loopholes and configuration errors that are frequently exploited by intruders.

2.3 Resource Controls

After applying its consistency checks, *sbox* applies resource limitations to the current process using the BSD-derived *setrlimit()* system call. Limits include the size of the CGI process, its resident (virtual) size, the number of file descriptors it can open, the size of the largest single file it can create, and the number of subprocesses it can spawn.

sbox uses both "hard" resource limits and "soft" ones. The soft limits, which can be adjusted upwards by the CGI script simply by calling *setrlimit()* itself, are set at low, stringent values by default. The hard limits, which once set cannot be increased during the lifetime of the process, use more liberal values. For example, the maximum file size that the user-supported CGI script has a soft limit of 100K, and a hard limit of 2 megabytes. These values can be adjusted at *sbox* compile time. The exception to this rule is the hard ceiling on core dumps, which is set to size zero. This prevents the user's CGI script from creating core files and closes various exploits that make use of core dumps to recover confidential information or to overwrite other files.

The net result of this design is that user-supported CGI scripts will, by default, be executed in an environment with strict resource controls. If a CGI script requires more of a particular resource than the soft limits provide, it can increase the resource up to the preset hard limit by calling *setrlimit()* itself. This design limits problems caused by resource hogging scripts written by naive users without unduly restricting the options of sophisticated users who need more resources than the soft limits allow.

In addition to setting resource limits, *sbox* also nices its own process to a priority of 10. This helps keep CGI scripts from becoming too much of a drain on a loaded system. Unlike *setrlimit()* values, a priority level, once increased, can never be decreased.

The priority level and the soft and hard limits on all system resources are set at *sbox* compile time. The system administrator can change the default values, or choose not to set a particular limit at all.

2.4 Changing the Root Directory

The crux of *sbox* security is its change-root function. If configured to do so, *sbox* will use the *chroot()* system call to change its root directory to some subdirectory enclosing the target CGI script. When the target CGI script runs, it will be unable to access parts of the filesystem outside the new root directory. This closes a large number of CGI exploits, including unauthorized access to the system password file, the modification of user's `.rhosts` files, the creation of hard links to system files in `/tmp`, and many more. It also provides a way to control exactly which system binaries and other resources that user-maintained CGI scripts have access to.

Administrator-configurable options determine how *sbox* chooses which directory to make the new root. In order for the target CGI script to be executed, it must live within the subdirectory selected for the new root. However, most CGI scripts will also need access to copies of system files such as interpreters and shared libraries in order to function correctly. Because it is inconvenient for the user to intermix his CGI scripts with system files, these files are usually stored in directories parallel to the directory that contains the target script. Another consideration is the user's "document root", the directory that contains his static HTML files. A number of popular CGI scripts, including guestbook scripts and page counters, require access to the user's HTML pages. In order for these scripts to work under the *sbox* system, the user's document root, or a portion of it at least, must also be located within the new root directory.

The locations of the new root directory and the target CGI script itself are controlled by the configuration variables ROOT and CGI_BIN respectively. Both variables are pathnames relative to the user's document root. A typical configuration will use the following values:

```
ROOT      ".."
CGI_BIN   "../cgi-bin"
```

This configuration tells *sbox* to look for the target CGI script inside a directory named *cgi-bin* on the same level as the user's document root directory. The new root directory will be the parent of both the *cgi-bin* directory and the user's document root. To see how this works in practice, consider a Web site in which user-supported directories are located in `/u/username/pub/html`, where "username" is substituted with the login name of the user. In Apache, this setup could be accomplished using the configuration directive **UserDir pub/html**.

A typical listing for `/u/username/pub` might look like the example shown in Table 1.

When *sbox* starts up, it determines the user's document root by looking at the Apache settings, which reveals the directory `/u/fred/pub/html`. It applies the CGI_BIN relative path, to give `/u/fred/pub/cgi-bin` as the directory in which to search for the CGI executable, and then applies the ROOT relative path to give `/u/fred/pub` as the directory that will become the new root. When *sbox* makes the *chroot()* call, `/u/fred/pub` becomes the top of the directory tree, creating a directory hierarchy with a structure similar to a Unix root filesystem. Files and directories above *pub*, which might include the user's private files, are off limits.

A drawback to this scheme is that it makes the user's entire document tree visible to his CGI scripts, which might not always be desirable. However a slight modification improves the scheme by making only a selected portion of the user's document tree visible. In this improved scheme, the Web server is configured so that the user's document tree is found, for example, in `/u/username/public_html`, and *sbox* is configured to change its root to a directory named *sbox* that is completely outside the *public_html* document tree:

```
ROOT      "../sbox"
CGI_BIN   "../sbox/cgi-bin"
```

For this configuration to work seamlessly, the user's directory should be set up something like this:

```
% ls -F /u/fred
public_html/                        doc root
public_html/sbox/->../sbox/html/    link
sbox/bin/
sbox/cgi-bin/                       scripts
sbox/etc/
sbox/lib/
sbox/html/
sbox/tmp/
...
```

The user's CGI scripts will now execute within the restricted *sbox* subdirectory and have no access, by default, to the user's HTML document tree. However the user can grant access to selected HTML documents by placing them into `public_html/sbox/`, which is connected via a symbolic link to `sbox/html/`. This allows CGI-accessible files to be accessed directly with a URL like this one:

http://www.site.com/~fred/sbox/index.html

while *sbox*-controlled CGI scripts are accessed with a URL like this one:

http://www.site.com/cgi-bin/sbox/~fred/guestbook

and CGI scripts that need to read or manipulate static HTML files are passed the additional path information in URLs like this one:

```
% ls /u/username/pub
total 10
drwxr-xr-x          2 fred   users   1024   Oct 23 06:27   bin/       system binaries
drwxr-xr-x          3 fred   users   1024   Oct 19 20:44   cgi-bin/   CGI scripts
drwxr-xr-x          2 fred   users   1024   Oct 12 16:59   dev/       device special files
drwxr-xr-x          2 fred   users   1024   Oct 19 17:57   etc/       configuration files
drwxr-xr-x          2 fred   users   1024   Oct 22 19:14   html/      HTML document root
drwxr-xr-x          3 fred   users   1024   Oct 19 20:35   lib/       shared libraries
drwxr-xr-x          2 fred   users   1024   Oct 23 05:48   tmp/       temporary files
```

Table 1: Typical directory listing for a user-supported Web directory

http://www.site.com/cgi-bin/sbox/~fred/ guestbook/html/index.html

If the Apache web server is being used, these URLs can be simplified significantly with URL rewriting rules. An example of this is shown below.

2.5 Environment Cleansing

Before executing the target CGI script, *sbox* sets up a clean environment to run the target in. Depending on how the Web server was launched, there may be residual information in the environment that is not germaine to the CGI protocol or may in fact divulge sensitive information, such as database authentication information, or private PATH directories.

sbox filters the current environment, allowing through only those environment variables that are specified by the CGI/1.1 protocol, such as REMOTE_ADDR, or which contain fields from the incoming HTTP request header, such as HTTP_USER_AGENT. In addition, *sbox* recognizes and permits a small number of common extensions to the CGI/1.1 protocol, such as the DOCUMENT_ROOT and SERVER_ADMIN variables.

Other variables are not automatically copied into the target script's environment. In particular the PATH environment variable, because of its history of exploitation is **not** passed through. Instead PATH is set up using a constant "safe path" set at compile time. By default, the safe path is `/bin:/usr/bin:/usr/local/bin`. Because the target script will be running in a change-root directory, it is likely that only */bin* will be available to the target script.

When possible, *sbox* adjusts path-related environment variables so that they correctly reflect the change-rooted filesystem seen by the user's CGI scripts. Among the environment variables that are adjusted are the DOCUMENT_ROOT variable, which should point to the top of the user's document tree and PATH_TRANSLATED, which points to the file passed to the user's CGI script as additional path information.

2.6 Logging

Before passing control to the user's CGI script, *sbox* logs its actions. It prints out a timestamp, the name of the CGI script being executed, and the UID and GID of the process that it will execute the script as. Diagnostic information is also logged when *sbox*'s consistency checks fail, or when an error occurs during the processing or execution of the target CGI script.

By default, *sbox* sends its log entries to standard error, which on most web servers becomes incorporated into the shared server error log file. However *sbox* can instead be configured to write entries into a private log file. There's there's a performance penalty in keeping a private log file, since *sbox* must open the file for appending every time it runs.

The main rationale for having a log entry for each CGI script executed is that it provides an audit trail in the case of a CGI-based attack. The time of the attack can be correlated with the *sbox* log, and possibly lead to the identification of the script that was exploited. The *sbox* log could also be used to monitor CGI script usage for patterns suggestive of probing activity.

3 Practical Considerations

Configuring the *sbox* executable and preparing user-supported directories are the most tedious parts of using the *sbox* system. In order to reduce dependencies on the external environment, *sbox* does not use a configuration file. Instead, all its operational parameters are determined at compile time via a series of preprocessor `#defines`. About three dozen defines are contained in a single include file, `sbox.h`, which the system administrator must edit before compiling the executable. Fortunately, the vast majority of the defines are boilerplate values which will not need to be changed by most sites. Only about a half dozen are truly site-specific.

System administrators used to modern configuration scripts will probably be disappointed by this primitive configuration process, even though it is simple and straightforward. For this reason, a GNU *configure* style configuration script [Friesenhalm] is currently in preparation.

A more onerous task is setting up user-supported directories so that their CGI scripts run correctly in a change-root environment. On most modern Unices, compiled programs need one or more shared libraries in order to execute. Either the user's CGI scripts must be compiled statically, or the new root directory must contain a `/lib` subdirectory (or the dialect's equivalent) containing the shared libraries the user needs.

Other system support files may needed as well. CGI scripts that require access to the DNS system for hostname resolution will need an `/etc` subdirectory containing `resolv.conf`. Scripts that perform time calculations may need access to the compiled timezone file, `/usr/lib/zoneinfo/localtime`. Programs that need access to device special files, such as `/dev/null` and `/dev/zero` will need the appropriate files created with the **mknod** program. Scripts written in interpreted languages such as Perl will require a `/bin` directory containing the interpreter executable, and any support files that the interpreter needs, such as code libraries.

Clearly there are drawbacks to replicating a good chunk of the root filesystem for each user-supported web directory. For one thing, the disk storage requirements may become prohibitive on a system with many users. One solution is to limit the type of CGI scripts that users can write to a particular development system, such as Perl. Then only those files needed to support the Perl interpreter will have to be copied into the user's scripting directory.

Another solution to this problem is to use NFS to mount a trimmed set of `/lib`, `/bin`, and `/etc` directories in each user-supported directory. Even after the *chroot()* operation, the contents of these directories will continue to remain available to the user's CGI scripts. Although this technique creates a lot of mount points, the overhead for unused NFS mounts is minimal [Stern], and an automount daemon can be further used to reduce the load [Crosby]. However if this technique is used, care must be taken not to mount directories that contain sensitive information, such as an `/etc` directory that contains a live password file. This would defeat the purpose of the change-root system.

A minor drawback to using *sbox* is that it is not completely transparent to the user. Instead of writing natural-looking CGI URLs, users have to be trained to interpose `/cgi-bin/sbox` in front of any URL that points to a CGI script. On Apache servers, an elegant solution to this problem is to use the *mod_rewrite* URL rewriting module to automatically add the `/cgi-bin/sbox` prefix to users' CGI URLs.

For example, one could use a *mod_rewrite* URL rewrite rule to transform URLs of the form:

/~fred/cgi/guestbook

into URLs of the form:

/cgi-bin/sbox/~fred/guestbook

by adding these directives to Apache's configuration file:

```
RewriteEngine On
RewriteRule ^/~(.+)/cgi/(.+) \
            /cgi-bin/sbox/~$1/$2  [PT,NS]
```

Neither the remote user nor the the script's author ever sees the longer URL. The name transformation is completely transparent. As a bonus, this rewrite expression also correctly handles additional path information appended to the end of the URL.

In order to perform its *suid()*, *sgid()*, and *chroot()* functions, *sbox* must run with superuser privileges. This means that, like *cgiwrap* and *suEXEC*, it must be installed set-user-id to root. This fact should give any cautious Unix system administrator pause. However, *sbox* consists of only 700 lines of C code, all of which are available for public scrutiny. *sbox* is careful to avoid using static buffers and string copy operations that could cause a buffer overflow. It also checks its environment at startup time to confirm that it was invoked by the web server and not some other local user.

4 Conclusions

The *sbox* wrapper increases the security of web sites that need to run untrusted CGI scripts. It prevents different users' CGI scripts from interfering with each other by running each user's program under distinct user and group IDs. It prevents user-maintained scripts from accessing sensitive parts of the file system by running each script in a change-root directory. It lessens the impact of denial of service attacks by establishing per-process resource limits, and it avoids certain common misconfigurations by checking the environment for consistency before it launches the target CGI script. Lastly, it creates an audit trail that can be used to track down malicious or poorly implemented CGI scripts.

sbox is not a panacea for CGI woes. There are a variety of CGI-based attacks that *sbox* cannot prevent. Chief among these are network-based attacks. For example, if a CGI script can be tricked into probing a firewall system from within the protected network, there is nothing that *sbox* can do to prevent this type of attack. To completely insulate the user's environment from that of the host, you need to step out of the Unix domain and use a partitioned operating system, such as Hewlett Packward's VirtualVault technology [Hewlett Packard].

Finally, it is important to remember that the *sbox* wrapper alone won't make a Web site secure. CGI script precautions are just one component of a carefully considered site security policy that includes attention to operating system security, web server configuration, operating and backup procedures, and user education. While nothing is ever going to completely eliminate the risk of running untrusted CGI scripts on a Web server, the *sbox* wrapper does go a long way towards limiting the potential damage that poorly-written or malicious scripts can inflict.

5 Acknowledgments

Many thanks to Nathan Neulinger for the original *cgiwrap* program which inspired this work. I also wish to thank the members of the Apache Project, whose web server has proven that open source projects can provide the same power and stability as conventional products – if not more so.

6 Availability

sbox is written in ANSI C and compiles on multiple flavors of Unix. It can be used and redistributed freely. The complete package is available for download at

http://stein.cshl.org/WWW/software/sbox/

References

[Apache Group] The Apache Group (1998). *Apache suEXEC Support*, *http://www.apache.org/docs/suexec.html*

[Coar] Coar K and Robinson D (1998). *The WWW Common Gateway Interface, Version 1.1* Internet Draft. *ftp://ftp.ietf.org/internet-drafts/draft-coar-cgi-v11-00.txt*.

[CERT] Computer Emergency Response Team (CERT) (1996-1998). Multiple CGI-related advisories. *ftp://ftp.cert.org/pub/cert˙advisories/*

[Crosby] Crosby M (1997). *AMD - AutoMount Daemon.* The Linux Journal 35, March 1997. *http://www.ssc.com/LJ/issue35/amd.html.*

[Fowler] Fowler G. (1993). *The Shell as a Service.* Usenix Summer 1993 Technical Conference Proceedings, Cincinnati, Ohio. *http://www.usenix.org/publications/library*

[Friesenhalm] Friesenhalm B (1997). *Autoconf Makes for Portable Software.* Byte, November 1997.

[Hewlett Packard] Hewlett Packard Inc. (1998). *HP Internet Security VirtualVault Home Page. http://www.hp.com/security/products/virtualvault/*

[Garfinkle] Garfinkle S with Spafford G (1997). *Web Security & Commerce.*, O'Reilly & Associates, Sebastopol CA.

[Microsoft] Microsoft Corporation (1998). FrontPage 98 - Overview. *http://www.microsoft.com/products/prodref/571˙ov.htm.*

[Neulinger] Neulinger N (1996-1998). *cgiwrap.* http://www.umr.edu/~cgiwrap/

[Rubin] Rubin A, Geer D, and Ranum M (1997). Web Security Sourcebook. *John Wiley & Sons, New York.*

[Stein97] Stein L, How to Set Up and Maintain a Web Site, *Chapters 8-9. Addison-Wesley Longman, Boston.*

[Stein98a] Stein L (1998). Web Security: A Step-by-Step Reference Guide. *Addison Wesley Longman, Boston.*

[Stein98b] Stein L (1998). The Offical Guide to Programming with CGI.pm. *John Wiley & Sons, New York.*

[Stein98c] Stein L (1998). The Web Security FAQ. http://www.w3.org/Security/Faq

[Stern] Stern H (1991). Managing NFS and NIS. *O'Reilly & Associates, Sebastopol, CA.*

The MultiSpace: an Evolutionary Platform for Infrastructural Services

Steven D. Gribble, Matt Welsh, Eric A. Brewer, and David Culler
The University of California at Berkeley
{gribble,mdw,brewer,culler}@cs.berkeley.edu

Abstract

This paper presents the architecture for a *Base*, a clustered environment for building and executing highly available, scalable, but flexible and adaptable infrastructure services. Our architecture has three organizing principles: addressing all of the difficult service fault-tolerance, availability, and consistency problems in a carefully controlled environment, building that environment out of a collection of execution environments that are receptive to mobile code, and using dynamically generated code to introduce run-time-generated levels of indirection separating clients from services. We present a prototype Java implementation of a Base called the *MultiSpace*, and talk about two applications written on this prototype: the Ninja Jukebox (a cluster based music warehouse), and Keiretsu (an instant messaging service that supports heterogeneous clients). We show that the MultiSpace implementation successfully reduces the complexity of implementing services, and that the platform is conducive to rapid service evolution.

1 Introduction

```
The performance and utility of a personal
computer will be defined less by faster Intel
processors and new Microsoft software and
increasingly by Internet services and software.
```

c|net news article excerpt, 11/25/98

Once a disorganized collection of data repositories and web pages, the Internet has become a landscape populated with rich, industrial-strength applications. Many businesses and organizations have counterparts on the web: banks, restaurants, stock trading services, communities, and even governments and countries. These applications possess similar properties to traditional utilities such as the telephone network or power grid: they support large and potentially rapidly growing populations, they must be available 24x7, and they must abstract complex engineering behind simple interfaces. We believe that the Internet is evolving towards a service-oriented infrastructure, in which these high quality utility-like applications will be commonplace. Unlike traditional utilities, Internet services tend to rapidly evolve, are typically customizable by the end-user, and may even be composable.

Although today's Internet services are mature, the process of erecting and modifying services is quite immature. Most authors of complex, new services are forced to engineer substantial amounts of custom, service-specific code, largely because of the diversity in the requirements of each service—it is difficult to conceive of a general-purpose, reusable, shrink-wrapped, adequately customizable and extensible service construction product.

Faced with a seemingly inevitable engineering task, authors tend to adopt one of two strategies for adding new services to the Internet landscape:

Inflexible, highly tuned, hand-constructed services: by far, this is the most dominant service construction strategy found on the Internet. Here, service authors carefully design a system targeted towards a specific application and feature set, operating system, and hardware platform. Examples of such systems are large, carrier-class web search engines, portals, and application-specific web sites such as news, stock trading, and shopping sites. The rationale for this approach is sound: it leads to robust and high-performance services. However, the software architectures of these systems are too restrictive; they result in a fixed service that performs a single, rigid function. The large amount of carefully crafted and hand-tuned code means that these services are difficult to evolve; consider, for example, how hard it would be to radically change the behavior of a popular search engine service, or to move the service into a new environment—these sorts of modifications would take massive engineering effort.

"Emergent services" in a world of distributed objects: this strategy is just beginning to become popularized with architectures such as Sun's JINI [31] and the ongoing CORBA effort [25]. In this world, instead of erecting complex, inflexible services, large numbers of components or objects are made available over the wide area, and services emerge through the composition of many such components. This approach has the benefit that adding to the Internet landscape is a much simpler task, since the granularity of contributed components is much smaller. Because of the explicit decomposition of the world into much smaller pieces, it is also simpler to retask or extend services by dropping one set of components and linking in others.

There are significant disadvantages to this approach. As a side-effect of the more evolutionary nature of services, it is difficult to manage the state of the system, as state may be arbitrarily replicated and distributed across the wide area. Wide-area network partitions are commonplace, meaning that it is nearly impossible to provide consistency guarantees while maintaining a reasonable amount of system availability. Furthermore, although it is possible to make incremental, localized changes to the system, it is difficult to make large, global changes because the system components may span many administrative domains.

In this paper, we advocate a third approach. We argue that we can reap many of the benefits of the distributed objects approach while avoiding difficult state management problems by encapsulating services and service state in a carefully controlled environment called a *Base*. To the outside world, a Base provides the appearance and guarantees of a non-distributed, robust, highly-available, high-performance service. Within a Base, services aren't constructed out of brittle, restrictive software architectures, but instead are "grown" out multiple, smaller, reusable components distributed across a workstation cluster [3]. These components may be replicated across many nodes in the cluster for the purposes of fault tolerance and high performance. The Base provides the glue that binds the components together, keeping the state of replicated objects consistent, ensuring that all of the constituent components are available, and distributing traffic across the components in the cluster as necessary.

The rest of this paper discusses the design principles that we advocate for the architecture of a Base (section 2), and presents a preliminary Base implementation called the Ninja MultiSpace[1] (section 3) that uses techniques such as dynamic code generation and code mobility as mechanisms for demonstrating and evaluating our hypotheses of service flexibility, rapid evolution, and robustness under change. While we have begun preliminary explorations into the scalability and high availability aspects of our prototype, that has not been the explicit focus of this initial implementation, and instead remains the subject of future work. Two example services running on our prototype Base are described in section 4. In section 5, we discuss some of the lessons we learned while building our prototype. Section 6 presents related work, and in Section 7 we draw conclusions.

[1]The MultiSpace implementation is available with the Ninja platform release - see `http://ninja.cs.berkeley.edu`.

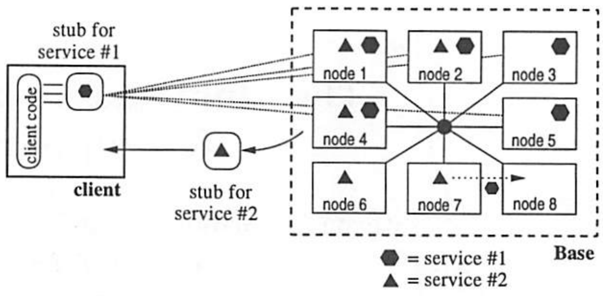


Figure 1: **Architecture of a Base:** a Base is comprised of a cluster of workstations connected by a high-speed network. Each node houses an execution environment into which code can be pushed. Services have many instances within the cluster, but clients are shielded from this by a service "stub" through which they interact with the cluster.

2 Organizing Principles

In this section we present three design principles that guided our service architecture development (shown at a high level in figure 1):

1. Solve the challenging service availability and scalability problems in carefully **controlled environments (Bases)**,
2. Gain service flexibility by decomposing Bases into a number of **receptive execution environments**, and
3. Introduce a level of indirection between the clients and services through the use of **dynamic code generation techniques**.

We now discuss each of these principles in turn.

2.1 Solve Challenging Problems in a Base

The high availability and scalability "utility" requirements that Internet services require are difficult to deliver; our first principle is an attempt to simplify the problem of meeting them by carefully choosing the environment in which we tackle these issues. As in [11], we argue that clusters of workstations provide the best platform on which to build Internet services. Clusters allow incremental scalability through the addition of extra nodes, high availability through replication and failover, and cost-performance by using commodity building blocks as the basis of the computing environment.

Clusters are the backbone of a *Base*. Physically, a Base must include everything necessary to keep a mission-critical cluster running: system administrators, a physically secure machine room, redundant internal networks and external network feeds, UPS systems, and so on. Logically, a cluster-wide software layer provides data consistency, availability, and fault tolerance mechanisms.

The power of locating services inside a Base arises from the assumptions that service authors can now make when designing their services. Communication is fast and local, and network partitions are exceptionally rare. Individual nodes can be forced to be as homogeneous as necessary, and if a node dies, there will always be an identical replacement available. Storage is local, cheap, plentiful, and well-guarded. Finally, everything is under a single domain, simplifying administration.

Nothing outside of the Base should try to duplicate the fault-tolerance or data consistency guarantees of the Base. For example, e-mail clients should not attempt to keep local copies of mail messages except in the capacity of a cache; all messages are permanently kept by an e-mail service in the Base. Because services promise to be highly available, a user can rely on being able to access her email through it while she is network connected.

2.2 Receptive Execution Environments

Internet services are generally built from a complex assortment of resources, including heterogeneous single-CPU and multiprocessor systems, disk arrays, and networks. In many cases these services are constructed by rigidly placing functionality on particular systems and statically partitioning resources and state. This approach represents the view that a service's design and implementation are "sanctified" and must be carefully planned and laid out across the available hardware. In such a regime, there is little tolerance for failures which disrupt the balance and structure of the service architecture.

To alleviate the problems associated with this approach, the Base architecture employs the principle of *receptive execution environments*—systems which can be dynamically configured to host a component of the service software. A collection of receptive execution environments can be constructed either from a set of homogeneous workstations or more diverse resources as required by the service. The distinguishing feature of a receptive execution environment, however, is that the service is "grown" on top of a fertile platform; functionality is *pushed into* each node as appropriate for the application. Each node in the Base can be remotely and dynamically configured by uploading service code components as needed, allowing us to delay the decision about the details of a particular node's specialization as far as possible into the service construction and maintenance lifecycle.[2]

As we will see in section 3, our approach has been to make a single assumption of homogeneity across systems in a Base: a Java Virtual Machine is available on each node. In doing so, we raise the bar of service construction by providing a common instruction set across all nodes, unified views on threading models, underlying system APIs (such as socket and filesystem access), as well as the usual strong typing and safety features afforded by the Java environment. Because of these provisions, any service component can be pushed into any node in our Base and be expected to execute, subject to local resource considerations (such as whether a particular node has access to a disk array or a CD drive). Assuming that every node is capable of receiving Java bytecodes, however, means that techniques generally applied to mobile code systems [21, 13, 28, 32, 18] can be employed internally to the Base: the administrator can deploy service components by uploading Java classes into nodes as needed, and the service can push itself towards resources redistributing code amongst the participating nodes. Furthermore, because in this environment we are restricting our use of code mobility to deploying local code within the scope of a single, trusted administrative domain, some of the security difficulties of mobile code are reduced.

[2]We rely on two mechanisms for mobile code security:we restrict the use of mobile code inside the Base to code that originates from trusted sources within the Base itself, and we use the Java Security Manager mechanism to sandbox this mobile code. Our research goals, however, do not include solving the mobile code security problem.

2.3 Dynamic Redirector Stub Generation

One challenge for clustered servers is to present a single service interface to the outside world, and to mask load-balancing and failover mechanisms in the cluster. The naive solution is to have a single front-end machine that clients first contact; the front-end then dispatches these incoming requests to one of several back-end machines. Failures can be hidden through the selection of another back-end machine, and load-balancing can be directly controlled by the front-end's dispatch algorithm. Unfortunately, the front-end can become a performance bottleneck and a single point of failure [11, 8]. One solution to these problems is to use multiple front-end machines, but this introduces new problems of naming (how clients determine which front-end to use) and consistency (whether the front-ends mutually agree on the back-end state of the cluster). The naming problem can be addressed in a number of ways, such as round-robin DNS [6], static assignment of front-ends to clients, or "lightweight" redirection in the style of scalable Web servers [17]. The consistency problem can be solved through one of many distributed systems techniques [4] or ignored if consistent state is unimportant to the front-end nodes.

The Base architecture takes another approach to cluster access indirection: the use of *dynamically-generated Redirector Stubs.* A stub is client-side code which provides access to a service; a common example is the stub code generated for CORBA/IIOP [25] and Java Remote Method Invocation (RMI) [23] systems. The stub code runs on the client and converts client requests for service functionality (such as Java method calls) into network messages, marshalling request parameters and unmarshalling results. In the case of Java RMI, clients download stubs on-demand from the server.

Base services employ a similar technique to RPC stub generation except that the Redirector Stub for a service is dynamically generated at run-time and contains embedded logic to select from a set of nodes within the cluster (figure 2). Load balancing is implemented within this "Redirector Stub", and failover is accomplished by reissuing failed or timed-out service calls to an alternative back-end machine. The redirection logic and information about the state of the Base is built up by the Base and advertised to clients periodically; clients obtain the Redirector Stubs from a registry. This methodology has a number of significant implications about the nature of services, namely that they must be idempotent and maintain self-consistency across a

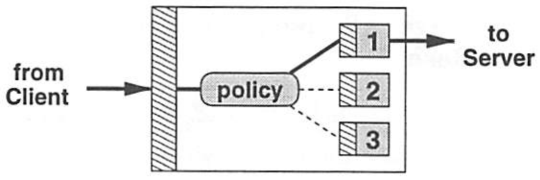


Figure 2: **A "Redirector Stub":** embedded inside a Redirectory Stub are several RPC stubs with the same interface, each of which communicates with a different service instance inside a Base.

service's instances on different nodes in the Base. In section 3.3.1, we will discuss an implementation of a cluster-wide distributed data structure that simplifies the task of satisfying these implications.

Client applications can be coded *without knowledge of the Redirector Stub logic*; by moving failover and load-balancing functionality to the client, the use of front-end machines can be avoided altogether. This is similar to the notion of smart clients [34], but with the intelligence being injected into the client at run-time instead of being compiled in.

3 Implementation

Our prototype Base implementation (written in Java) is called the *MultiSpace*. It serves to demonstrate the effectiveness of our architecture in terms of facilitating the construction of flexible services, and to allow us to begin explorations into the issues of our platform's scalability. The MultiSpace implementation has three layers: the bottom layer is a set of communications primitives (NinjaRMI); the middle layer is a single-node execution environment (the iSpace); and the top layer is a set of multiple node abstractions (the MultiSpace layer). We describe each of these in turn, from the bottom up.

3.1 NinjaRMI

A rich set of high performance communications primitives is a necessary component of any clustered environment. We chose to make heavy use of Java's Remote Method Invocation (RMI) facilities for performing RPC-like [5] calls across nodes in the cluster, and between clients and services. When a caller invokes an RMI method, stub code intercepts the invocation, marshalls arguments, and sends them to a remote "skeleton" method handler for unmarshalling and execution. Using RMI as the finest granularity communication data unit in our clustered environment has many useful properties. Because method invocations have completely encapsulated, atomic semantics, retransmissions or communication failures are easy to reason about—they correspond to either successful or failed method invocations, rather than partial data transmissions.

However, from the point of view of clients, if a remote method invocation does not successfully return, it can be impossible for the client to know whether or not the method was successfully invoked on the server. The client has two choices: it can reinvoke the method call (and risk calling the same method twice), or it can assume that the method was not invoked, risking that the method was in fact invoked, successfully or unsuccessfully with an exception, but results were not returned to the client. Currently, on failure our Redirector Stubs will retry using a different, randomly chosen service stub; in the case of many successive failures, the Redirector Stub will return an exception to the caller. It is because of the at-least-once semantics implied by these client-side reinvocations that we must require services to be idempotent. The exploration of different retry policies inside the Redirector Stubs is an area of future research.

3.1.1 NinjaRMI enhancements to Sun's RMI

NinjaRMI is a ground-up reimplementation of Sun's Java Remote Method Invocation for use by components within the Ninja system. NinjaRMI was designed to permit maximum flexibility in implementation options. NinjaRMI provides three interesting transport-level RMI enhancements. First, it provides a unicast, UDP-based RMI that allows clients to call methods with "best-effort" semantics. If the UDP packet containing the marshalled arguments successfully arrives at the service, the method is invoked; if not, no retransmissions are attempted. Because of this, we enforce the requirement that such methods do not have any return values. This transport is useful for beacons, log entries, or other such side-effect oriented uses that do not require reliability. Our second enhancement is a multicast version of this unreliable transport. RMI services can associate themselves with a multicast group, and RMI calls into that multicast group result in method invocations on all listening services. Our third enhancement is to provide very flexible certificate-based authentication and encryption support for reliable, unicast RMI. Endpoints in an RMI session can associate themselves with digital certificates issued by a certification authority. When the TCP-connection underlying an RMI session is established, these certificates are exchanged by the RMI layer and verified at each endpoint. If the certificate verification succeeds, the remainder of the communication over that TCP connection is encrypted using a Triple DES session key obtained from a Diffie-Hellman key exchange. These security enhancements are described in [12].

Packaged along with our NinjaRMI implementation is an interface compiler which, when given an object that exports an RMI interface, generates the client-side stub and server-side skeleton stubs for that object. All source code for stubs and skeletons can be generated dynamically at run-time, allowing the Ninja system to leverage the use of an intelligent code-generation step when constructing wrappers for service components. We use this to introduce the level of indirection needed to implement Redirector Stubs—stubs can be overloaded to cause method invocations to occur on many remote nodes, or for method invocations to fail over to auxiliary nodes in the case of a primary node's failure.

3.1.2 Measurements of NinjaRMI

Method	Local method	Sun RMI	Ninja RMI
`f(void)`	0.19 μs	0.83 ms	0.82 ms
`f(int)`	0.20 μs	0.84 ms	0.85 ms
`int f(int)`	0.18 μs	0.85 ms	0.84 ms
`int f( int,int,int,int)`	0.22 μs	0.88 ms	0.86 ms
`f(byte[100])`	0.19 μs	1.06 ms	1.05 ms
`f(byte[1000])`	0.20 μs	1.20 ms	1.09 ms
`f(byte[10000])`	0.21 μs	2.21 ms	2.23 ms
`byte[100] f(int)`	0.19 μs	1.07 ms	1.00 ms
`byte[1000] f(int)`	0.20 μs	1.17 ms	1.01 ms
`byte[10000] f(int)`	0.20 μs	2.20 ms	2.10 ms

Table 1: **NinjaRMI microbenchmarks:** These benchmarks were gathered on two 400Mhz Pentium II based machines, each with 128MB of physical memory, connected by a switched 100 Mb/s Ethernet, and using Sun's JDK 1.1.6v2 with the TYA just-in-time compiler on Linux 2.0.36. For the sake of comparison, UDP round-trip times between two C programs were measured at 0.185 ms, and between two Java programs at 0.316 ms.

As shown in table 1, NinjaRMI performs as well as or better than Sun's Java RMI package. Given that a null RMI invocation cost 0.82 ms and that a round-trip across the network and through the JVMs cost 0.316 ms, we conclude that the difference (roughly 0.5 ms) is RMI marshalling and protocol overhead. Profiling the code shows that the main contributor to this overhead is object serialization, specifically the use of methods such as `java.io.ObjectInputStream.read()`.

3.2 iSpace

A Base consists of a number of workstations, each running a suitable receptive execution environment for single-node service components. In our prototype, this receptive execution environment is the *iSpace*: a Java Virtual Machine (JVM) that runs a component loading service into which Java classes can be pushed as needed. The iSpace is responsible for managing component resources, naming, protection, and security. The iSpace exports the component loader interface via NinjaRMI; this interface allows a remote client to obtain a list of components running on the iSpace, obtain an RMI stub to a particular component, and upload a new service component or kill a component already running on the iSpace (subject to authentication). Service components running on the iSpace are protected from one another and from the surrounding execution environment in three ways:

1. each component is coded as a Java class which provides protection from hard crashes (such as null pointer dereferences),
2. components are separated into thread groups; this limits the interaction one component can have with threads of another, and
3. all components are subject to the iSpace Security Manager, which traps certain Java API calls and determines whether the component has the credentials to perform the operation in question, such as file or network access.

Other assumptions must be made in order to make this approach viable. In essence, we are relying on the JVM to behave and perform like a miniature operating system, even though it was not designed as such. For example, the Java Virtual Machine does not provide adequate protection between threads of multiple components running within the same JVM: one component could, for example, consume the entire CPU by running indefinitely within a non-blocking thread. Here, we must assume that the JVM employs a preemptive thread scheduler (as is true in the Sun's Solaris Java environment) and that fairness can be guaranteed through its use. Likewise, the iSpace Security Manager must utilize a strategy for resource management which ensures both fairness and safety. In this respect iSpace has similar goals to other systems which provide multiple protection domains within a single JVM, such as the JKernel [16]. However, our approach does not necessitate a re-engineering of the Java runtime libraries, particularly because intra-JVM thread communication is not a high-priority feature.

3.3 MultiSpace

The highest layer in our implementation is the MultiSpace layer, which tethers together a collection of iSpaces (figure 3). A primary function of this layer is to provide each iSpace with a replicated registry of all service instances running in the cluster.

A MultiSpace service inherits from an abstract MultiSpaceService class, whose constructor registers each service instance with a "MultiSpaceLoader" running on the local iSpace. All MultiSpaceLoaders in a cluster cooperate to maintain the replicated registry; each one periodically sends out a multicast beacon[3] that carries its list of local services to all other nodes in the MultiSpace, and also listens for multicast messages from other MultiSpace nodes. Each MultiSpaceLoader builds an independent version of the registry from these beacons. The registries are thus soft-state, similar in nature to the cluster state maintained in [11] and [1]—if an iSpace node goes down and comes back up, its MultiSpaceLoader simply has to listen to the multicast channel to rebuild its state. Registries on different nodes may see temporary periods of inconsistency as services are pushed into the MultiSpace or moved across nodes in the MultiSpace, but in steady state, all nodes asymptotically approach consistency. This consistency model is similar in nature to that of Grapevine [10].

The multicast beacons also carry RMI stubs for each local service component, which implies that any service instance running on any node in the MultiSpace can identify and contact any other service in the MultiSpace. It also means that the MultiSpaceLoader on every node has enough information to construct Redirector Stubs for all of the services in the MultiSpace, and advertise those Redirector Stubs to off-cluster clients through a "service discovery service" [9].[4] Each RMI stub embedded in

[3]Currently, IP multicast is used for this purpose — the multicast channel that beacons are sent over thus defines the logical scope and boundary of an individual MultiSpace. We intend to replace this transport with multicast NinjaRMI.

[4]The service discovery service (or SDS) implementation consists of an XML search engine that allows client programs to locate services based on arbitrary XML predicates.

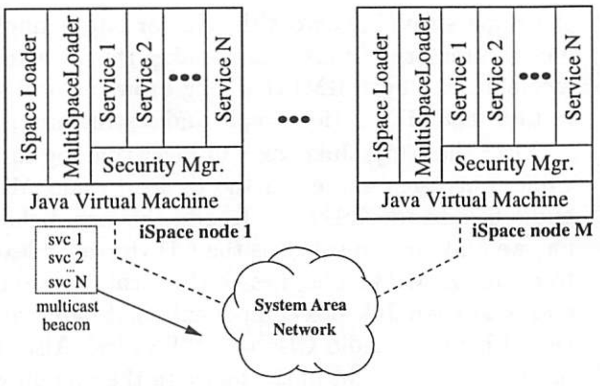


Figure 3: **The MultiSpace implementation:** MultiSpace services are instantiated on top of a sandbox (the Security Manager), and run inside the context of a Java virtual machine (JVM).

a multicast beacon is (based on our observations) roughly 500 bytes in average length; there is therefore an important tradeoff between the beacon frequency (and therefore freshness of information in the MultiSpaceLoaders) and the number of services whose stubs are being beaconed that will ultimately affect the scalability of the MultiSpace.

Service instances can elect to receive multicast beacons from other MultiSpace nodes; a service can use this mechanism to become aware of its peers running elsewhere in the cluster. If a service overrides the standard beacon class, it can augment its beacons with additional information, such as the load that it is currently experiencing. Services that want to call out to other services could thus make coarse grained load balancing decisions without requiring a centralized load manager.

3.3.1 Built-In MultiSpace Services

Included with the MultiSpace implementation are two services that enrich the functionality available to other MultiSpace services: the distributed hash table, and the uptime monitoring service.

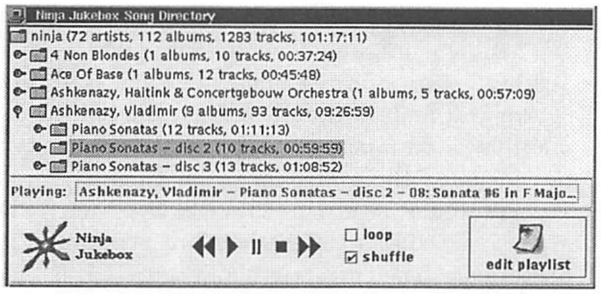
Distributed hash table: as mentioned in section 2.3, due to the Redirector Stub mechanism MultiSpace service instances must maintain self-consistency across the nodes of the cluster. To make this task simpler, we have provided service authors with a distributed, replicated, fault-tolerant hash table that is implemented in C for the sake of efficiency. The hash table is designed to present a consistent view of data across all nodes in the cluster, and as such, services may use it to rendezvous in a style similar to Linda [2] or IBM's T-Spaces [33]. The current implementation is moderately fast (it can handle more than 1000 insertions per second of 500 byte entries on a 4 node 100Mb/s MultiSpace cluster), is fault tolerant, and transparently mask multiple node failures. However, it does not yet provide all of the consistency guarantees that some services would ideally prefer, such as on-line recovery of crashed state or transactions across multiple operations. The currently implementation is, however, suitable for many Internet-style services for which this level of consistency is not essential.

Uptime monitoring service: Even with the service beaconing mechanism, it is difficult to detect the failure of individual nodes in the cluster. This is partly because of the lack of a clear failure model in Java: a Java service is just a collection of objects, not necessarily even possessing a thread of execution. A service failure may just imply that a set of objects has entered a mutually inconsistent state. The absence of beacons doesn't necessarily mean that a service instance failure has occurred; the beacons may be lost due to congestion in the internal network, or the beacons may not have been generated because the service instance is overloaded and is busy processing other tasks.

For this reason, we have provided an uptime monitoring abstraction to service authors. If a service running in the MultiSpace implements a well-known Java interface, the infrastructure automatically detects this and begins periodically calling the `doProbe()` method in that interface. By implementing this method, service authors promise to perform an application-level task that demonstrates that the service is accepting and successfully processing requests. By using this application-level uptime check, the infrastructure can explicitly detect when a service instance has failed. Currently, we only log this failure in order to generate uptime statistics, and we rely on the Redirector Stub failover mechanisms to mask these failures.

4 Applications

In this section of the paper, we discuss two applications that demonstrate the validity of our guiding principles and the efficacy of our MultiSpace implementation. The first application, the Ninja Jukebox, abstracts the many independent compact-disc players and local filesystems in the Berkeley Network of Workstations (NOW) cluster into a single pool of available music. The second, Keiretsu, is a three-tiered application that provides instant messaging across heterogeneous devices.



```
Ninja Jukebox Song Directory
ninja (72 artists, 112 albums, 1283 tracks, 101:17:11)
  4 Non Blondes (1 albums, 10 tracks, 00:37:24)
  Ace Of Base (1 albums, 12 tracks, 00:45:48)
  Ashkenazy, Haitink & Concertgebouw Orchestra (1 albums, 5 tracks, 00:57:09)
  Ashkenazy, Vladimir (9 albums, 93 tracks, 09:26:59)
    Piano Sonatas (12 tracks, 01:11:13)
    Piano Sonatas - disc 2 (10 tracks, 00:59:59)
    Piano Sonatas - disc 3 (13 tracks, 01:08:52)
Playing: Ashkenazy, Vladimir - Piano Sonatas - disc 2 - 08: Sonata #6 in F Majo...
```

Figure 4: **The Ninja Jukebox GUI:** users are presented with a single Jukebox interface, even though songs in the Jukebox are scattered across multiple workstations, and may be either MP3 files on a local filesystem, or audio CDs in CD-ROM drives.

4.1 The Ninja Jukebox

The original goal of the Ninja jukebox was to harness all of the audio CD players in the Berkeley NOW (a 100+ node cluster of Sun UltraSparc workstations) to provide a single, giant virtual music jukebox to the Berkeley CS graduate students. The most interesting features of the Ninja Jukebox arise from its implementation on top of iSpace: new nodes can be dynamically harnessed by pushing appropriate CD track "ripper" services onto them, and the features of the Ninja Jukebox are simple to evolve and customize, as evidenced by the seamless transformation of the service to the batch conversion of audio CDs to MP3 format, and the authenticated transmission of these MP3s over the network.

The Ninja Jukebox service is decomposed into three components: a master directory, a CD "ripper" and indexer, and a gateway to the online CDDB service [22] that provides artist and track title information given a CD serial number. The ability to push code around the cluster to grow the service proved to be exceptionally useful, since we didn't have to decide a priori which nodes in the cluster would house CDs—we could dynamically push the ripper/indexer component towards the CDs as the CDs were inserted into nodes in the cluster. When a new CD is added to a node in the NOW cluster, the master directory service pushes an instance of the ripper service into the iSpace resident on that node. The ripper scans the CD to determine what music is on it. It then contacts a local instance of the CDDB service to gather detailed information about the CD's artist and track titles; this information is put into a playlist which is periodically sent to the master directory service. The master directory incorporates playlists from all of the rippers running across the cluster into a single, global directory of music, and makes this directory available over both RMI (for song browsing and selection) and HTTP (for simple audio streaming).

After the Ninja Jukebox had been running for a while, our users expressed the desire to add MP3 audio files to the Jukebox. To add this new behavior, we only had to subclass the CD ripper/indexer to recognize MP3 audio files on the local filesystem, and create new Jukebox components that batch converted between audio CDs and MP3 files. Also, to protect the copyright of the music in the system we added access control lists to the MP3 repositories, with the policy that users could only listen to music that they had added to the Jukebox.[5] We then began pushing this new subclass to nodes in the system, and our system evolved while it was running.

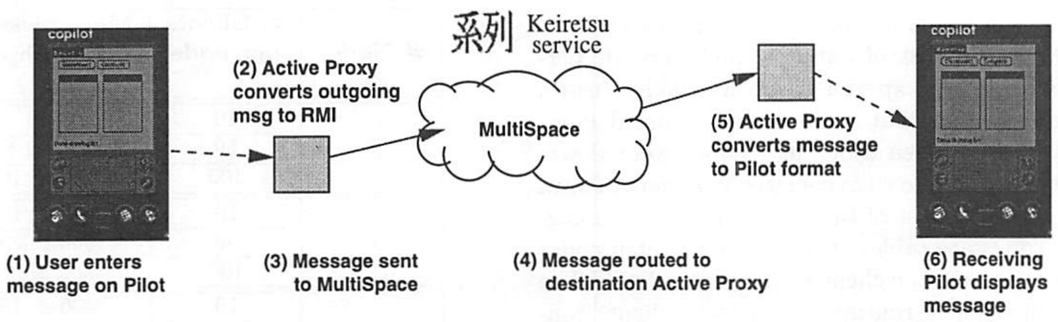
The performance of the Ninja Jukebox is completely dominated by the overhead of authentication and the network bandwidth consumed by streaming MP3 files. The first factor (authentication overhead) is currently benchmarked at a crippling 10 seconds per certificate exchange, entirely due to a pure Java implementation of a public key cryptosystem. The second factor (network consumption) is not quite as crippling, but still significant: each MP3 consumes at least 128 Kb/s, and since the MP3 files are streamed over HTTP, each transmission is characterized by a large burst as the MP3 is pushed over the network as quickly as possible. Both limitations can be remedied with significant engineering, but this would be beyond the scope of our research.

4.2 Keiretsu: The Ninja Instant-Messaging Service

Keiretsu[6] is a MultiSpace service that provides instant messaging between heterogeneous devices: Web browsers, one- or two-way pagers, and PDAs such as the Palm Pilot (see Figure 5). Users are able to view a list of other users connected to the Keiretsu service, and can send short text messages to other users. The service component of Keiretsu exploits the MultiSpace features: Keiretsu service instances use the soft-state registry of peer nodes in order to exchange client routing information across the cluster, and automatically generated Redirector Stubs are handed out to clients for use in communicating with Keiretsu nodes.

[5]This ACL policy is enforced using the authentication extensions to NinjaRMI described in 3.1.1.

[6]*Keiretsu* is a Japanese concept in which a group of related companies work together for each other's mutual success.



系列 Keiretsu service
copilot
(2) Active Proxy converts outgoing msg to RMI
MultiSpace
(5) Active Proxy converts message to Pilot format
copilot
(1) User enters message on Pilot
(3) Message sent to MultiSpace
(4) Message routed to destination Active Proxy
(6) Receiving Pilot displays message

Figure 5: **The Keiretsu Service**

Keiretsu is a three-tired application: simple client devices (such as pagers or Palm Pilots) that cannot run a JVM connect to an Active Proxy, which can be thought of as a simplified iSpace node meant to run soft-state mobile code. The Active Proxy converts simple text messages from devices into NinjaRMI calls into the Keiretsu MultiSpace service. The Active Proxies are assumed to have enough sophistication to run Java-based mobile code (the protocol conversion routines) and speak NinjaRMI, while rudimentary client devices need only speak a simple text-based protocol.

As described in section 2.3, Redirector Stubs are used to access the back-end service components within a MultiSpace by pushing load-balancing and failover logic towards the client—in the case of simple clients, Redirector Stubs execute in the Active Proxy. For each protocol message received by an Active Proxy from a user device (such as "send message M to user U"), the Redirector Stub is invoked to call into the MultiSpace.

Because the Keiretsu proxy is itself a mobile Java component that runs on an iSpace, the Keiretsu proxy service can be pushed into appropriate locations on demand, making it easy to bootstrap such an Active Proxy as needed. State management inside the Active Proxy is much simpler than state management inside a Base—the only state that Active Proxies maintain is the session state for connected clients. This session state is soft-state, and it does not need to be carefully guarded, as it can be regenerated given sufficient intelligence in the Base, or by having users manually recover their sessions.

Rudimentary devices are not the only allowable members of a Keiretsu. More complex clients that can run a JVM speak directly to the Keiretsu, instead of going through an Active Proxy. An example of such a client is our e-mail agent, which attaches itself to the Keiretsu and acts as a gateway, relaying Keiretsu messages to users over Internet e-mail.

4.2.1 The Keiretsu MultiSpace service

```
public void identifySelf(
  String clientName,
  KeiretsuClientIF clientStub);

public void disconnectSelf(String clientName);

public void injectMessage(KeiretsuMessage msg);

public String[] getClientList();
```

Figure 6: **The Keiretsu service API**

The MultiSpace service that performs message routing is surprisingly simple. Figure 6 shows the API exported by the service to clients. Through the `identifySelf` method, a client periodically announces its presence to the Keiretsu, and hands the Keiretsu an RMI stub which the service will use to send it messages. If a client stops calling this method, the Keiretsu assumes the client has disconnected; in this way, participation in the Keiretsu is treated as a lease. Alternately, a client can invalidate its binding immediately by calling the `disconnectSelf` method. Messages are sent by calling the `injectMessage` method, and clients can obtain a list of other connected clients by calling the `getClientList` method.

Inside the Keiretsu, all nodes maintain a soft-state table of other nodes by listening to MultiSpace beacons, as discussed in section 3.3. When a client connects to a Keiretsu node, that node sends the client's RMI stub to all other nodes; all Keiretsu nodes maintain individual tables of these client bindings. This means that in steady state, each node can route messages to any client.

Because clients access the Keiretsu service through Redirector Stubs, and because Keiretsu nodes replicate service state, individual nodes in the

Keiretsu can fail and service will continue uninterrupted, at the cost of capacity and perhaps performance. In an experiment on a 4-node cluster, we demonstrated that the service continued uninterrupted even when 3 of the 4 nodes went down. The Keiretsu source code consists of 5 pages of Java code; however, most of the code deals with managing the soft-state tables of the other Keiretsu nodes in the cluster and the client RMI stub bindings. The actual business of routing messages to clients consists of only *half a page of Java code*—the rest of the service functionality (namely, building and advertising Redirector Stubs, tracking service implementations across the cluster, and load balancing and failover across nodes) is hidden inside the MultiSpace layer. We believe that the MultiSpace implementation is quite successful in shielding service authors from a significant amount of complexity.

4.2.2 Keiretsu Performance

We ran an experiment to measure the performance and scalability of our MultiSpace implementation and the Keiretsu service. We used a cluster of 400 MHz Pentium II machines, each with 128 MB of physical memory, connected by a 100 Mb/s switched Ethernet. We implemented two Keiretsu clients: the "speedometer", which open up a parameterizable number of identities in the Keiretsu and then waits to receive messages, and the "driver", which grabs a parameterizable number of Redirector Stubs to the Keiretsu, downloads a list of clients in the Keiretsu, and then blasts 75 byte messages to randomly selected clients as fast as it can.

We started our Keiretsu service on a single node, and incrementally grew the cluster to 4 nodes, measuring the maximum message throughput obtained for $10x$, $50x$, and $100x$ "speedometer" receivers, where x is the number of nodes in the cluster. To achieve maximum throughput, we added incrementally more "driver" connections until message delivery saturated. The drivers and speedometers were located on many dedicated machines, connected to the Keiretsu cluster by the same 100 Mb/s switched Ethernet. Table 2 shows our results.

For a small number of receivers (10 per node), we observed linear scaling in the message throughput. This is because each node in the Keiretsu is essentially independent: only a small amount of state is shared (the client stubs for the 10 receivers per node). In this case, the CPU was the bottleneck, likely due to Java overhead in message processing and argument marshalling and unmarshalling.

For larger number of receivers, we observed a

# Nodes	# Clients per node	Max. message throughput (msgs / s)
1	10	246 ± 4
	50	200 ± 8
	100	195 ± 10
2	10	420 ± 10
	50	300 ± 20
	100	260 ± 20
3	10	490 ± 15
	50	370 ± 20
	100	160 ± 15
4	10	570 ± 15
	50	210 ± 10
	100	120 ± 10

Table 2: **Keiretsu performance:** These benchmarks were run on 400Mhz Pentium II machines, each with 128MB of physical memory, connected by a 100 Mb/s switched Ethernet, using Sun's JDK 1.1.6v2 with the TYA just-in-time compiler on Linux 2.2.1, and sending 75 byte Keiretsu messages.

breakdown in scaling when the total number of receivers reached roughly 200 (i.e. 3-4 nodes at 50 receivers per node, or 2 nodes at 100 receivers per node). The CPU was still the bottleneck in these cases, but most of the CPU time was spent processing the client stubs exchanged between Keiretsu nodes, rather than processing the clients' messages. This is due to poor design of the Keiretsu service; we did not need to exhange client stubs as frequently as we did, and we should have simply exchanged timestamps for previously distributed stubs rather than repeatedly sending the same stub. This limitation could be also removed by modifying the Keiretsu service to inject client stubs in a distributed hash table, and rely on service instances to pull the stubs out of the table as needed. However, a similar N^2 state exchange happens at the MultiSpace layer with the multicast exchange of service instance stubs; this could potentially become another scaling bottleneck for large clusters.

5 Discussion and Future Work

Our MultiSpace and service implementation efforts have given some insights into our original design principles, and into the use of Java as an Internet service construction language. In this section of the paper, we delve into some of these insights.

5.1 Code Mobility as a Service Construction Primitive

When we were designing the MultiSpace, we knew that code mobility would be a powerful tool. We originally intended to use code mobility for delivering code to clients (which we do in the form of Redirector Stubs), and for it to be used by clients to upload customization code into the MultiSpace (which has not been implemented). However, code mobility turned out to be useful *inside* the MultiSpace as a mechanisms for structuring services, and distributing service components across the cluster.

Code mobility solved a software distribution problem in the Jukebox, without us realizing that software distribution might become a problem. When we updated the ripper service, we needed to distribute the new functionality to all of the nodes in the cluster that would potentially have CD's inserted into them. Code mobility also partially solved the service location problem in the Jukebox: the ripper services depend on the CDDB service to gather detailed track and album information, but the ripper has no easy way to know where in the cluster the CDDB service is running. Using code mobility to push the CDDB service onto the same node as the ripper, we enforced the invariant that the CDDB service is colocated with the ripper.

5.2 Bases are a Simplifying Principle, but not a Complete Solution

The principle of solving complex service problems in a Base makes it easier to reason about the interactions between services and clients, and to ensure that difficult tasks like state management are dealt with in an environment in which there is a chance of success. However, this organizational principle alone is not enough to solve the problem of constructing highly available services. Given the controlled environment of a Base, service authors must still construct services to ensure consistency and availability. We believe that a Base can provide primitives that further simply service authors' jobs; the Redirector Stub is an example of such a primitive.

While building the Ninja Jukebox and Keiretsu, we made observations about how we achieved availability and consistency. Most of the code in these services dealt with distributing and maintaining tables of shared state. In the Ninja Jukebox, this state was the list of available music. In Keiretsu, this state was the list of other Keiretsu nodes, and the tables of client stub bindings. The distributed hash table was not yet complete when these two services were being implemented. If we had relied on it instead of our ad-hoc peer state exchange mechanisms, much of the services' source code would have been eliminated.

In both of these services, work queues are not explicitly exposed to service authors. These queues are hidden inside the thread scheduler, since NinjaRMI spawns a thread per connected client. This design decision had repercussions on service structure: each service had to be written to handle multithreading, since service authors must handle consistency within a single service instance as well as across instances throughout the cluster. Providing a mechanism to expose work queues to service authors may simplify the structure of some services, for example if services serialize requests to avoid issues associated with multithreading.

In the current MultiSpace, service instances must explicitly keep track of their counterparts on other nodes and spawn new services when load or availability demands it. A useful primitive would be to allow authors to specify conditions that dictate when service instances are spawned or pushed to a specific node, and to allow the MultiSpace infrastructure to handle the triggering of these conditions.

5.3 Java as an Internet-service construction environment

Java has proven to be an invaluable tool in the development of the Ninja infrastructure. The ability to rapidly deploy cross-platform code components simply by assuming the existence of a Java Virtual Machine made it easy to construct complex distributed services without concerning oneself with the heterogeneity of the systems involved. The use of RMI as a strongly-typed RPC, tied very closely to the Java language semantics, makes distributed programming comparably simple to single node development. The protection, modularization, and safety guarantees provided by the Java runtime environment make dynamic dissemination of code components a natural activity. Similarly, the use of Java class reflection to generate new code wrappers for existing components (as with Redirector Stubs) provides automatic indirection at the object level.

Java has a number of drawbacks in its current form, however. Performance is always an issue, and work on just-in-time [14, 24] and ahead-of-time [27] compilation is addressing many of these problems. The widely-used JVM from Sun Microsystems exhibits a large memory footprint (we have observed 3-4 MB for "Hello World", and up to 30

MB for a relatively simple application that performs many of memory allocations and deallocations[7]), and crossing the boundary from Java to native code remains an expensive operation. In addition, the Java threading model permits threads to be non-preemptive, which has serious implications for components which must run in a protected environment. Our approach has been to use only Java Virtual Machines which employ preemptive threads.

We have started an effort to improve the performance of the Java runtime environment. Our initial prototype, called Jaguar, permits direct Java access to hardware resources through the use of a modified just-in-time compiler. Rather than going through the relatively expensive Java Native Interface for access to devices, Jaguar generates machine code for direct hardware access which is inlined with compiled Java bytecodes. We have implemented a Jaguar interface to a VIA[8] enabled fast system area network, obtaining performance equivalent to VIA access from C (80 microseconds round-trip time for small messages and over 400 megabits/second peak bandwidth). We believe that this approach is a viable way to tailor the Java environment for high-performance use in a clustered environment.

6 Related Work

Directly related to the Base architecture is the TACC [11] platform, which provides a cluster-based environment for scalable Internet services. In TACC, service components (or "workers") can be written in a number of languages and are controlled by a front-end machine which dispatches incoming requests to back-end cluster machines, incorporating load-balancing and restart in the case of node failure. TACC workers may be chained across the cluster for composable tasks. TACC was designed to support Internet services which perform data transformation and aggregation tasks. Base services can additionally implement long-lived and persistent services; the result is that the Ninja approach addresses a wider set of potential applications and system-support issues. Furthermore, Base services can dynamically created and destroyed through the iSpace loader interface on each MultiSpace node—TACC did not have this functionality.

[7]There are no explicit deallocations in Java — by "deallocation", we mean discarding all references to an object, thus enabling it to be garbage collected.

[8]VIA is the Virtual Interface Architecture [26], which specifies an industry-standard architecture for high-bandwidth, low-latency communication within clusters.

Sun's JINI [31] architecture is similar to the the Base architecture in that it proposes to develop a Java-based *lingua franca* for binding users, devices, and services together in an intelligent, programmable Internet-wide infrastructure. JINI's use of RMI as the basic communication substrate and use of code mobility for distributing service and device interfaces has a great deal of rapport with our approach. However, we believe that we are addressing problems which JINI does not directly solve: providing a hardware and software environment supporting scalable, fault-tolerant services is not within the JINI problem domain, nor is the use of dynamically-generated code components to act as interfaces into services. However, JINI has touched on issues such as service discovery and naming which have not yet been dealt with by the Base architecture; likewise, JINI's use of JavaSpaces and "leases" as a persistent storage model may interact well with the Base service model.

ANTS [30, 29] is a system that enables the dynamic deployment of mobile code which implements network protocols within Active Routers. Coded in Java, and utilizing techniques similar to those in the iSpace environment, ANTS has a similar set of goals in mind as the Base architecture. However, ANTS uses mobile code for processing each packet passing through a router; Base service components are executed on the granularity of an RMI call. Liquid Software [19] and Joust [15] are similar to ANTS in that they propose an environment which uses mobile code to customize nodes in the network for communications oriented tasks. These systems focused on adapting systems at the level of network protocol code, while the Base architecture uses code mobility for distribution of service components both internally to and externally from a Base.

SanFrancisco [7] is a platform for building distributed, object-oriented business applications. Its primary goal is to simplify the development of these applications by providing developers a set of industrial-strength "Foundation" objects which implement common functionality. As such, SanFrancisco is very similar to Sun's Enterprise Java Beans in that it provides a framework for constructing applications using reusable components, with SanFrancisco providing a number of generic components to start with. MultiSpace addresses a different set of goals than Enterprise Java Beans and SanFrancisco in that it defines a flexible runtime environment for services, and MultiSpace intends to provide scalability and fault-tolerance by leveraging the flexibility of a component architecture. MultiSpace services could be built using the EJB or SanFrancisco model (extended to expose the MultiSpace functionality), but these issues appear to be orthogonal.

The Distributed Computing Environment (DCE) [20] is a software suite that provides a middleware platform that operates on many operating systems and environments. DCE abstracts away many OS and network services (such as threads, security, a directory service, and RPC) and therefore allows programmers to implement DCE middleware independent of the vagaries of particular operating systems. DCE is rich, robust, but notoriously heavyweight, and its focus is on providing interoperable, wide-area middleware. MultiSpace is far less mature, but focuses instead on providing a platform for rapidly adaptable services that are housed within a single administrative domain (the Base).

7 Conclusions

In this paper, we presented an architecture for a *Base*, a clustered hardware and software platform for building and executing flexible and adaptable infrastructure services. The Base architecture was designed to adhere to three organizing principles: **(1)** solve the challenging service availability and scalability problems in carefully *controlled environments*, allowing service authors to make many assumptions that would not otherwise be valid in an uncontrolled or wide area environment, **(2)** gain service flexibility by decomposing Bases into a number of *receptive execution environments*; and **(3)** introduce a level of indirection between the clients and services through the use of *dynamic code generation techniques*.

We built a prototype implementation (MultiSpace) of the Base architecture using Java as a foundation. This implementation took advantage of the code mobility and dynamic compilation techniques to help in the structuring and deployment of services inside the cluster. The MultiSpace abstracts away load balancing, failover, and service instance detection and naming from service authors. Using the MultiSpace platform, we implemented two novel services: the Ninja Jukebox, and Keiretsu. The Ninja Jukebox implementation demonstrated that code mobility is valuable inside a cluster environment, as it permits rapid evolution of services, and run-time binding of service components to available resources in the cluster. The Keiretsu application demonstrated that our MultiSpace layer successfully reduced the complexity of building new services: the core Keiretsu service functionality was implemented in less than a page of code, but the application was demonstrably fault-tolerant. We also demonstrated that Keiretsu code limited the scalability of this service, rather than any inherent limitation in the MultiSpace layer, although we hypothesized that our use of multicast beacons would ultimately limit the scalability of the current MultiSpace implementation.

8 Acknowledgements

The authors thank Ian Goldberg and David Wagner for their contributions to iSpace, NinjaRMI, and the Jukebox application. We would also like to thank all of the graduate and undergraduate students in the Ninja project for agreeing to be guinea pigs for our platform, and Brian Shiratsuki and Keith Sklower for their help in the system of the MultiSpace machines and networks. Finally, we would like to thank our anonymous reviewers and our shepherd, Charles Antonelli, for their insight and suggestions for improvement.

References

[1] Elan Amir, Steven McCanne, and Randy Katz. An Active Service Framework and its Application to Real-Time Multimedia Transcoding. In *Proceedings of ACM SIGCOMM '98*, volume 28, pages 178–189, October 1998.

[2] B. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query Processing. In *Springer-Verlag Lecture Notes in Computer Science 574*, Mont-Saint-Michel, France, June 1991.

[3] Thomas E. Anderson, David E. Culler, and David Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 12(1):54–64, February 1995.

[4] Ken Birman, Andre Schiper, and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[5] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Call. *ACM Transactions on Computing Systems*, 2(1):39–59, February 1984.

[6] T. Brisco. RFC 1764: DNS Support for Load Balancing, April 1995.

[7] IBM Corporation. IBM SanFrancisco product homepage. `http://www.software.ibm.com/ad/sanfrancisco/`.

[8] Inktomi Corporation. The Technology Behind HotBot. `http://www.inktomi.com/whitepap.html`, May 1996.

[9] Steven Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony Joseph, and Randy Katz. An Architecture

for a Secure Service Discovery Service. In *Proceedings of MobiCom '99*, Seattle, WA, August 1999. ACM.

[10] A.D. Birrell et al. Grapevine: An Exercise in Distributed Computing. *Communications of the Association for Computing Machinery*, 25(4):3–23, Feb 1984.

[11] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

[12] Ian Goldberg, Steven D. Gribble, David Wagner, and Eric A. Brewer. The Ninja Jukebox. In *Submitted to the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, Colorado, USA, October 1999.

[13] Robert S. Gray. Agent Tcl: A Flexible and Secure Mobile-Agent System. In *Proceedings of the Fourth Annual Usenix Tcl/Tk Workshop*. USENIX Association, 1996.

[14] The Open Group. The Fajita Compiler Project. `http://www.gr.opengroup.org/java/compiler/fajita/index-b.htm`, 1998.

[15] J. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proebsting, and O. Spatscheck. Joust: A Platform for Liquid Software. In *IEEE Network (Special Edition on Active and Programmable Networks)*, July 1998.

[16] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing Multiple Protection Domains in Java. In *Proceedings of the 1998 Usenix Annual Technical Conference*, June 1998.

[17] Open Group Research Institute. Scalable Highly Available Web Server Project (SHAWS). `http://www.osf.org/RI/PubProjPgs/SFTWWW.htm`.

[18] Dag Johansen, Robbert van Renesse, , and Fred R. Schneider. Operating System Support for Mobile Agents. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, 1995.

[19] John Hartman and Udi Manber and Larry Peterson and Todd Proebsting. Liquid Software: A New Paradigm for Networked Systems. Technical report, Department of Computer Science, University of Arizona, June 1996.

[20] Brad Curtis Johnson. A Distributed Computing Environment Framework: an OSF Perspective. Technical Report DEV-DCE-TP6-1, the Open Group, June 1991.

[21] Eric Jul, Henry M. Levy, Norman C. Hutchinson, and Andrew P. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computing Systems*, 6(1):109–133, 1988.

[22] Ti Kan and Steve Scherf. CDDB Specificaton. `http://www.cddb.com/ftp/cddb-docs/cddb_howto.gz`.

[23] Sun Microsystems. Java Remote Method Invocation—Distributed Computing for Java. `http://java.sun.com/`.

[24] Sun Microsystems. The Solaris JIT Compiler. `http://www.sun.com/solaris/jit`, 1998.

[25] The Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification, February 1998. `http://www.omg.org/library/c2indx.html`.

[26] Virtual Interface Architecture Organization. Virtual Interface Architecture Specification version 1.0, December 1997. `http://www.viarch.org`.

[27] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for Applications—A Way Ahead of Time (WAT) Compiler. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies (COOTS)*, Portland, Oregon, USA, June 1997.

[28] Joseph Tardo and Luis Valente. Mobile Agent Security and Telescript. In *Proceedings of the 41st International Conference of the IEEE Computer Society (CompCon '96)*, February 1996.

[29] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A Survey of Active Network Research. Active Networks home page (MIT Telemedia, Networks and Systems group), 1996.

[30] David L. Tennenhouse and David J. Wetherall. Towards an Active Network Architecture. In *ACM SIGCOMM '96 (Computer Communications Review)*. ACM, 1996.

[31] Jim Waldo. Jini Architecture Overview. Available at `http://java.sun.com/products/jini/whitepapers`.

[32] James E. White. Telescript Technology: The Foundation for the Electronic Marketplace, 1991. `http://www.generalmagic.com`.

[33] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. TSpaces. *IBM Systems Journal*, 37(3), April 1998.

[34] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of the Winter 1997 USENIX Technical Conference*, January 1997.

Web++: A System For Fast and Reliable Web Service

Radek Vingralek[a]
Yuri Breitbart

Information Science Research Center
Bell Laboratories - Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974
{rvingral,yuri}@research.bell-labs.com

Mehmet Sayal
Peter Scheuermann

Northwestern University
ECE Department
2145 Sheridan Avenue
Evanston, IL 60208
{mehmet,peters}@ece.nwu.edu

[a]Current affiliation: STAR Lab, InterTrust Technologies, 460 Oakmead Parkway, Sunnyvale, CA 94086, rvingral@intertrust.com.

Abstract

We describe the design of a system for a fast and reliable HTTP service termed Web++. Web++ achieves high reliability by dynamically replicating Web data among multiple Web servers. Web++ selects a server which is available and that is expected to provide the fastest response time. Furthermore, Web++ guarantees data delivery, provided that at least one server containing the requested data is available. After detecting a server failure, Web++ client requests are satisfied transparently to the client by another server. Web++ is built on top of the standard HTTP protocol and does not require any changes either in existing Web browsers, or the installation of any software on the client side. We implement a Web++ prototype; performance experiments indicate that Web++ improves the client response time on average by 36.6%, and in many cases by as much as 59%, when compared with the current Web performance.

1 Introduction

1.1 Motivation

The success of the Web has proven the value of sharing different types of data in an autonomous manner. The number of Web users, servers, and total Internet traffic have been growing exponentially in the past 5 years [1]. The scale of Web usage is stressing the capacity of the Internet infrastructure and leads to poor performance and low reliability of Web service. Multisecond response times for downloading a 1KB resource are not unusual [35]. Furthermore, recent studies [29] indicate that server mean time to failure (MTTF) is 15 days, thus a client accessing 10 servers may experience a failure every 36.4 hours. Such a failure rate is not acceptable for many important Web applications such as electronic commerce and online stock trading. Recently, several techniques have been adopted to reduce Web response time, improve its reliability, and balance load among Web servers. Among the most popular approaches are:

Proxy server caching. Proxy servers intercept client requests and cache frequently referenced data. Requests are intercepted either at an application protocol level (*non-transparent caches*) [20, 30] or a network protocol level (*transparent caches*) [13]. Caching improves the response time of subsequent requests that can be satisfied directly from the proxy cache.

Server clusters A single dispatcher intercepts all Web requests and redirects them to one of the servers in the cluster. The requests are intercepted at the network protocol level [12, 16]. Since a server cluster typically is located within a single LAN, the server selection is mostly based on server load and availability within the cluster.

DNS aliasing A single host name is associated with multiple IP addresses. A modified DNS server selects one of the IP addresses based either on round-robin scheduling [26], routing distance, or TCP/IP probe response time [14].

Each of the proposed solutions, however, improves request response time, reliability, or load balancing among servers, but does not address all these issues together. Furthermore, many of the proposed solutions often introduce additional problems. Proxy server caching improves request response time, but it introduces potential data inconsistency between the cached data and the same data stored at the server. Non-transparent proxy caches create a single point of failure. Server clusters improve reliability at the server end, but do not address the reliability of the network path between the cluster dispatcher and a client. Server clusters are not suitable for balancing a load among geographically replicated Web servers because all requests must pass through a single dispatcher. Consequently, server clusters only improve the load balance among the back end Web servers. Finally, although DNS aliasing improves both request response time and service reliability, it forces data providers to replicate the entire Web site. This is impractical for two reasons: (1) since most Web servers exhibit skewed access pattern [32], replicating the entire Web server could be an overkill; (2) in some cases it is not possible or desirable to replicate all dynamic services. In addition, the DNS aliasing implementation becomes problematic when client-side DNS agents cache results of DNS queries or submit recursive DNS queries.

1.2 Paper Preview

One way to improve Web performance and reliability is to replicate popular Web resources among different servers. If one of the servers fails, clients satisfy their requests from other servers that contain replicas of the same resource. Client requests can be directed to the "closest" server that contains the requested resource and thereby improve the request response time. Replication also allows the balancing of clients' requests among different servers and enables "cost-conscious scalability" [9, 42] of the Web service whereby a surge in a server load can be handled by dynamically replicating *hot* data on additional servers.
In this paper we present an overview of design of our Web++ system for replication of the HTTP service. Unlike other similar systems reported in literature, Web++ is completely transparent to the browser user and requires no changes to the existing Web infrastructure. Web++ clients are downloaded as cryptographically signed applets to commercially available browsers. There is no need for end-users to install a plug-in or client-side proxy. There is no need for any modification of the browser; the Web++ applet can execute in both Netscape Navigator 4.x and Microsoft Explorer 4.x browsers. Web servers that support servlets can be directly extended with Web++ servlets. Other servers are extended with a server-side proxy that supports servlets. All client-to-server and server-to-server communication is carried on top of HTTP 1.1. Other salient features of Web++ are:

Reliability Resources are replicated among multiple Web++ servers. If one of the servers fails, clients transparently fail-over to another server that replicates the requested resource. After a failure repair, the server transparently returns to service without affecting clients. Furthermore, Web++ guarantees data delivery if at least one of the servers holding the requested resource is available.

Fast response time User's requests are directed by Web++ to the server that is expected to provide the fastest response time among all other available servers where the resource is replicated. This is done transparently to the user and the user is not required to know which server has delivered the resource.

Dynamic replication If there is a high demand for a resource, the resource can be dynamically replicated on another server that is lightly loaded or close to the clients that frequently request the resource. Furthermore, when demand for a resource drops, some servers may drop the resource copy. Additional servers may be recruited from a pool of underutilized servers to help sustain a load peak.

Light-Weight Clients The client applets maintain very little state information. In fact, the only information that they maintain is the HTTP latency of the various servers. This allows our system to be run on many hardware configurations with limited resources.

2 Web++ Architecture

The Web++ architecture is shown in Figure 1. It consists of Web++ clients and Web++ servers. Both client-to-server and server-to-server communication is carried on top of the standard HTTP 1.1 protocol [18]. Users submit their requests to Web++ client, which is a *Smart Client* [44], i.e., a standard Web browser (Netscape Navigator 4.x or Microsoft Internet Explorer 4.x) extended by downloading a cryptographically signed applet. The Web++ applet must be signed so that it can execute outside of the security "sandbox". The Web++ client sends user requests to a Web++ server, which is a standard HTTP server extended either with a Web++ servlet or a server-side proxy. The Web++ server returns the requested resource that can be either statically or dynamically generated.
Each Web++ resource contains a set of *logical URLs* to reference other resources. Before resource retrieval, each logical URL is bound to one of the *physical URLs* that corresponds to one of the resource replicas. A *physical URL* is the naming scheme currently used on the Web as specified in [6]. A *logical* URL is similar to Uniform Resource Name (URN) [39] in that it uniquely identifies a single resource independently of its location. Unlike the URN specification, there are no restrictions on syntax of logical URLs. In fact, a physical URL of resource replica can also be considered as a logical URL[1]. The only difference between a logical and a physical URL lies in their interpretation at resource retrieval time. While a physical URL is directly used to retrieve a resource, a logical URL first must be bound to a physical URL. The binding is done by the Web++ applet that executes within the user's browser.

[1]In Section 3.3 we explain why it may be useful to use one of the physical URLs as a logical URL.

After receiving a resource, the Web++ applet intercepts any events that are triggered either due to the browser's parsing of a resource or due to a user following logical URLs embedded in a retrieved resource. For each logical URL the applet finds a list of *physical URLs* that correspond to the resource's replicas. The list is embedded into the referencing resource by the Web++ servlet. Using the resource replica selection algorithm, the applet selects the physical URL that corresponds to a resource held by an available server that is expected to deliver the best response time for the client. If, after sending the request, the client does not receive a response, the applet fails over to the next fastest server. This process continues until the entire list of physical URLs is exhausted or the resource is successfully retrieved.

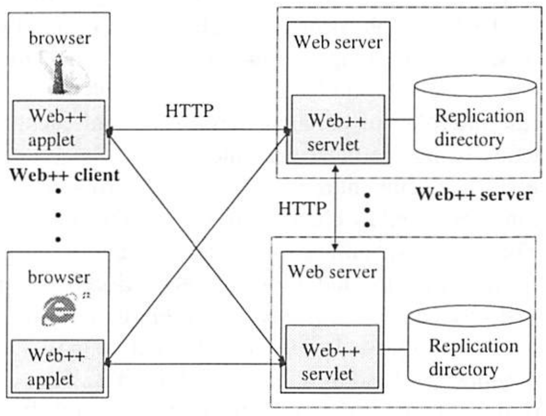


Figure 1: **Web++ architecture.**

Each server maintains a *replication directory* that maps each logical URL into a set of physical URLs that identify the locations of replicas of the resource. For example, the replication directory may contain an entry

```
/misc/file.txt :
http://server1.com/files/miscFile.txt
http://server2.com/misc/file.txt
```

that indicates that `/misc/file.txt` is a logical URL of a resource replicated on two servers, `server1.com` and `server2.com`. To reduce the size of the replication directory, the logical URL suffixes may use a wild card to specify replication locations for a set of resources.

Web++ servers are capable of creating, destroying and updating resource replicas. After creation or destruction of a replica each server automatically updates its local copy of the replication directory and propagates the update to other servers. The servers guarantee eventual consistency of their replication directories in the presence of concurrent updates propagated from different servers. Web++ servers also pre-process each HTML resource sent to a client by embedding physical URLs corresponding to each logical URL that occurs in the resource. The servers also embed a reference to the Web++ client applet within each HTML resource. Finally, Web++ servers also keep track of the load associated with their resources. The load statistics can be used by user-supplied policies that determine when, where and which resource should be replicated or when a replica should be destroyed.

3 Web++ Client Design

In this section we describe our Web++ client design and discuss the major design decisions. Web++ client binds logical URLs received in every HTML resource to one of the physical URLs corresponding to the replicas of the resources referenced by the logical URL. By binding a logical URL to a physical URL, the client selects the "closest" replica of a resource and fails over to another replica if the closest replica is not available. The current Web++ client implementation consists of approximately 18 KB of Java bytecode.

3.1 Location of Web++ client

The conversion from a logical URL into a physical URL can be done at several points on the path between a client and a server:

- Server or server-side proxy
- Client-side proxy
- Browser

A single server has only incomplete information about the network topology and cannot easily predict the kind of end-to-end performance a client would receive from other servers. Similarly, it is difficult for a server to predict whether the client would be able to reach any of the remaining servers. Therefore, the binding of logical URL to a physical URL should be done close to the client. This can be achieved by embedding the binding algorithm in a client-side proxy [4]. We see, however, several problems with the proxy approach. First, it requires making some changes to the existing proxies. Second, the proxy approach is a one-size-fits-all solution. For the same proxy, it is difficult for different content providers to use different algorithms for server selection. Third, the client-side proxy is inflexible for upgrades since it requires large numbers of users to install new versions or patches. Finally, adding a proxy on the path between a browser and a server leads to performance degradation. To quantify the performance degradation, we re-executed the client trace collected in a public lab at Northwestern University in a period of three days [35] and sent each request either directly to the Internet or via an Apache 1.2.6 proxy. To quantify only the proxy-access overhead, the proxy did no caching. The results in Figure 2 show that a commodity proxy may increase the response time by as much as 28% (223.2 ms). Similar results were obtained in [28]. We conjecture that the performance degradation is partly due to the store-and-forward implementation of Apache, i.e., the first byte of the response is not forwarded to the browser until the last byte of the body has been received by the proxy.

connectivity	average response time (ms)
direct	794.6
Apache	1017.8

Figure 2: **Proxy access overhead.**

Ideally, the client (or a client-size proxy) should dynamically download the code that performs the binding. The code can be downloaded together with the data. Such a solution does not require the end-user to install any software or to upgrade the browser. Different data providers may use different algorithms to perform the binding. Finally, upgrades can be quickly distributed among a majority of users. We are aware of two technologies satisfying the above criteria: Java applets and ActiveX controls. Since Java applets are supported by both Netscape Navigator and Microsoft Explorer browsers, we opted for an applet based implementation of Web++ client.
We also measured the overhead of executing an applet within a browser. Both Microsoft Internet Explorer 4.x and Netscape Navigator 4.x incur a relatively high overhead (3 s) to initialize the Java Virtual Machine. However, such initialization is done only once per browser session and could be done asynchronously (unfortunately, both browsers do the initialization synchronously upon parsing the first reference to an applet). We found that the execution of an applet method that implements the binding of a logical URL to a physical URL took on average 15 ms on Netscape Navigator 4.002 and 26 ms on Microsoft Internet Explorer 4.0[2]. In both cases the extra overhead is an order of magnitude smaller than the overhead incurred by using an Apache proxy and less than 4% of the average response time measured in the trace.
We observe that the Web++ applet does not have to be downloaded with every resource. In particular, the applet can be cached by the browser as any other resource. The default browser behavior is that the applet's timestamp is compared to that on the source server (using HTTP conditional `GET`) only once during each browser session.

3.2 Logical to Physical URL Binding

A given Web++ client applet must first find a list of physical URLs that correspond to replicas of every logical URL found in each HTML resource. The list can be found in several ways:

- The client queries a name server to get the list of physical URLs corresponding to a given logical URL. The name service can be either independent of the Web servers or some of the Web servers can also act as name servers. A scheme based on independent name servers similar to the DNS service was proposed for binding of Uniform Resource Names (URNs) to IP addresses [39].

- The server looks up all lists of physical URLs corresponding to every logical URL that occurs in a requested HTML resource. The list is piggybacked on the response sent to the client.

A drawback of the first scheme is that the client may have to incur an additional network round trip to bind a logical URL to a physical URL. The network round trip can be saved by caching the binding information on the client, but such a solution leads to several problems on its own (the interplay of dynamic replication and cache consistency being the most prominent one). Since one of the goals of document replication is to improve the response time perceived by clients, we rejected this option.
The second scheme does not lead to any extra overhead for the client to bind a logical URL. Moreover, since the majority of URL requests can be predicted from the hyperlinks embedded in the HTML text, it makes sense to optimize the binding scheme for this case. The drawback of this scheme is that it is not clear how to resolve logical URLs which are directly supplied by the end-user via e.g. `File` → `Open` browser menus.

3.3 Transfer of Control to an Applet

Every event that leads to sending an HTTP request (such as clicking on a hyperlink or parsing a reference to an embedded image) needs to be intercepted by the Web++ applet in order to bind the logical URL to one of the physical URLs embedded in the resource. We found two possible solutions:

- Let the browser itself render every resource along with the necessary graphical controls (e.g. "Back" button). Since the applet itself renders all graphical elements, it is simple to intercept all of the important events.

- Add JavaScript event handlers into the HTML source. The event handler transfers control to the Web++ applet when the events occur.

The first solution leads to a duplication of the browser's functionality within the Web++ applet. We rejected this solution because, in general, duplication of code is a poor software engineering practice. In this specific case, it

[2]Both browsers executed on a PC with 300 MHz Pentium II processor and 64 MB of main memory running Windows NT Workstation 4.0.

would be difficult to keep the applet rendering capabilities in sync with the latest version of HTML implemented by browsers. We therefore adopted the second approach that does not lead to any functionality duplication. However, due to the limitation of the `java.applet` API, not all important events can be intercepted by the applet. We discuss these cases below.

The HTML source modification is performed by the Web++ server. The server expands each reference to a logical URL in the resource (which typically follows <`HREF`> or <`SRC`> HTML tags) with an invocation of a JavaScript event handler. The event handler updates the value of the hyperlink reference when the hyperlink is clicked upon. For example, the hyperlink

```
        <A HREF="/misc/file.txt">
```

is replaced by the server with

```
<A HREF="/misc/file.txt"
onClick="this.ref =
document.Webpp.getUrl(
http://server1.com/files/miscFile.txt,
http://server2.com/misc/file.txt)">
```

References to embedded resources (following the <`SRC`> HTML tag) are expanded in a similar manner. On browsers that do not support JavaScript, the `onClick` event handler is ignored and the supplied URL is used instead. Therefore, it is beneficial to select the logical URL to correspond to one of the physical URLs, e.g. the physical URL of a primary copy.

The parameters passed to the applet method directly correspond to an entry in the replication directory of the server. The above method of including the list of physical URLs directly into the HTML source may lead to an increase of size of the resource that must be transmitted to the client. However, it is possible to put all the binding information into a new HTTP header, which is then read by the client applet. Standard compression techniques can be applied on the content of the header to reduce the size of the transmitted data. The compression may be particularly effective if many of the resources are replicated on the same set of servers leading to a high redundancy in the header content. We are currently in the process of implementing the above optimization and evaluating its impact on the performance.

3.4 Batch Resource Transmission

Having a client that can be downloaded directly into the browser creates many additional optimization opportunities of the HTTP protocol. For example, based on the response time perceived by a client, the server may compress not only the content of the header containing the URL binding information, but the entire resource. The resource is decompressed by the receiving client applet. Similarly, since the server substitutes all references to embedded resources (following the <`SRC`> HTML tag), it can transmit to the client not only the requested resource, but also all resources embedded in it in a single response (the embedded resources are typically located on the same server). The client must be able to exclude from transmission the resources that are already cached in its local cache.

Such a "batch transmission" leads to considerable savings since most commercial Web pages contain 20 to 40 embedded images. Following standard HTTP, the browser parses the containing HTML resource and sends separate GET request for each of the embedded resources. Most browsers reduce the total retrieval time by sending 4 to 5 requests in parallel and reusing the TCP connections [34]. However, even with such optimizations, downloading a typical Web page leads to at least 4 to 5 GET request rounds. The batch transmission method reduces the entire process into a single round with a large response.

Batch transmission is implemented in our Web++ prototype and we are in the process of evaluating its impact on the response time perceived by browser users as well as the number of IP packets transmitted. Our preliminary results indicate that for clients connected to Internet over a fast T3 line, batch loading can reduce the response time by additional 40% to 52% (depending on the number and size of the embedded resources and the distance between the client and server). We also found the saving is much smaller for clients connected over a 56 kbps modem and a phone line (between 13% and 16%), because such clients are mostly limited by the phone line bandwidth, and not by the communication latency.

3.5 Limitations of Java Applets

Our implementation of the Web++ applet revealed also several limitations of the `java.applet` API:

- Applets cannot stream data directly into the browser.
- Applets cannot subscribe to events detected by the browser that are triggered outside of the applet area. For example, applets cannot detect that a user followed a bookmark. Similarly, applets cannot detect that a user typed in a URL that should be followed.

Our implementation of Web++ client applet circumvents the first limitation by writing the received resource into a local file and passing its URL to the browser. Such a mechanism allows us to implement a local browser cache that matches resources based on their logical URL as opposed to physical URL matching used in most browsers. The second limitation could be addressed (although inefficiently) by re-implementing the necessary graphical controls (i.e. bookmark button) directly within the browser area.

Both of the limitations are eliminated in the ActiveX "Pluggable Protocol" interface that is supported by Microsoft Internet Explorer 4.x browser. We plan to explore a Web++ client implementation based on this technology as well as to investigate implementation of a similar interface within the publicly available Netscape Navigator source code.

4 Replica Selection Algorithms

The performance improvement achieved by using a replicated Web service, such as Web++, critically depends on the design of an algorithm that selects one of the replicas of the requested resource. The topic has been recently a subject of intensive study in the context of Internet services [11, 22, 23, 38, 17, 35, 27, 19]. Each of the replica selection algorithms can be described by the goals that should be achieved by replica selection, the metrics that are used for replica selection and finally, the mechanisms used for measuring the metrics. The replica selection algorithms may aim at maximizing network throughput [22, 19], reducing load on "expensive" links or reducing the response time perceived by the user [11, 38, 17, 35, 27]. Most replica selection algorithms aim at selection of "nearby" replicas to either reduce response time or the load on network links. The choice of a metric, which defines what are the "nearby" replicas, is crucial because it determines the effectiveness of achieving the goals of replica selection and also the overhead resulting from measurement of the metric. The metrics include response time[17, 27], latency [35], ping round-trip time[11], network bandwidth [38], number of hops [11, 22] or geographic proximity [23]. Since most of the above metrics are dynamic, replica selection algorithms typically rely on estimating the current value of the metric using samples collected in the past. The selected metric can be measured either actively by polling the servers holding the replicas [19] or passively by collecting information about previously sent requests [38] or a combination of both [17, 35].

4.1 The Extended Refresh Algorithm

The replica selection algorithm used in Web++ is an extension of the *Refresh* algorithm studied in [35]. The Web++ implementation of the Refresh algorithm extends the original algorithm in a number of ways:

- We extend the basic replica selection algorithm with support for fail-over.
- We reduce the size of the state maintained by the algorithm (i.e. the latency table described below) by using recursive formulas.
- We generalize the metric used for replica selection by using *percentiles*.

In addition, we also performed experiments in order to study the accuracy and stability of the estimates maintained by the algorithm. We first describe the basic features of the extended Refresh algorithm and justify their selection.

We chose to minimize the response time perceived by the end-user because this is the metric perceived by the end-user. Consequently, the HTTP request response time would be an ideal metric for selection of a "nearby" server. However, the response time depends also on resource size, which is unknown at the time of a request submission. Therefore, the HTTP request response time needs to be estimated using some other metric. We chose the HTTP request latency, i.e., the time to receive the first byte of the request, because we found that it is well correlated with the HTTP request response time as shown in Figure 3. The results in Figure 3 are based on client-side proxy traces collected in the computer lab of Northwestern University and further described in [35].

metric	correlation
#hops	0.16
ping RTT	0.51
HTTP latency	0.76

Figure 3: **Correlation with HTTP request response time.**

We chose a combination of active and passive measurement of HTTP request latency. Namely, most of the time clients passively reuse the statistics they collected from previously sent requests. However, periodically, clients actively poll some of the servers that have not been used for a long time. Each Web++ client applet collects statistics about the latencies observed for each server and keeps them in a *latency table*, which is persistently stored on a local disk. To increase the sampling frequency perceived by any individual client, the latency table is shared by multiple clients. In particular, the latency table is stored in a shared file system and is accessible to all clients using the file system[3]. We have implemented a rudimentary concurrency control mechanism to provide access to the shared latency table. Namely, the table is locked when clients synchronize their memory based copy with the disk based shared latency table. The concurrency control guarantees internal consistency of the table, but does not prevent lost updates. We believe that such a permissive concurrency control is adequate given that the latency table content is interpreted only as a statistical hint. The importance of sharing statistical data for clients using passive measurements has been pointed out in [38].

[3]If a shared file system is not available, each client uses its local version of latency table.

The estimate of the latency average, which is kept in the latency table, is used to predict the response time of a new request sent to a server. However, should two servers have similar average latencies, the latency variance should be used to break the tie, because it estimates the quality of service provided by a given server. There are several ways to combine the average and variance into a single metric. We chose a *percentile* because unlike e.g. statistical hypothesis testing it always provides an ordering among the alternatives.
An S-percentile is recursively estimated as

$$S\text{-}percentile = avg_{new} + \frac{c_S \cdot \sqrt{var_{new}}}{\sqrt{n}} \tag{1}$$

where S is the parameter that determines the percentile (such as 30, 50 or 85), avg_{new} is the current estimate of average, var_{new} is the current estimate of variance, c_S is an S-percentile of normal distribution (which is a constant) and n is the number of samples used for calculation of average and variance.
The average avg_{new} is estimated using a recursive formula

$$avg_{new} = (1 - r) \cdot avg_{old} + r \cdot sample \tag{2}$$

where avg_{new} and avg_{old} are new and old estimates of average, $sample$ is the current value of latency and r is a fine-tuning parameter. Similarly, the variance is estimated using [31]

$$var_{new} = (1-r) \cdot var_{old} + r \cdot (sample - avg_{new})^2 \tag{3}$$

where var_{new} and var_{old} are new and old estimates of variance.
The number of samples that affect the estimates in (2) and (3) continuously grows. Consequently, the importance of variance in (1) would decrease in time. However, the samples in (2) and (3) are exponentially weighted, so only a small fixed number of most recent samples affects the current estimates. Namely, the recursive formula for average (2) can be expanded as

$$avg_{new} = \sum_{k=1}^{N} r \cdot (1-r)^{N-k} sample_k + (1-r)^N sample_0 \tag{4}$$

where N is the total number of all samples and $sample_0$ is an initial estimate of the average. It is straightforward to derive from (4) that only the m most recent samples contribute to $100 \cdot p\%$ of the total weight where

$$m \geq \frac{\ln(1-p)}{\ln(1-r)} - 1 \tag{5}$$

Our extended Refresh algorithm selects the server with the minimum S-percentile of latency. Unfortunately, a straightforward implementation of a replica selection algorithm that selects resource replicas solely based on latencies of requests previously sent to the server holding the selected replica leads to a form of starvation. In particular, the replica selection is based on progressively more and more stale information about the replicas on servers that are not selected for serving requests. In fact, it has been shown that in many cases a random decision is better than a decision based on too old information [33]. There are several possibilities for implementing a mechanism for "refreshing" the latency information for the servers that have not been contacted in a long time. One possibility is to make the selection probabilistic, where the probability that a replica is selected is inversely proportional to HTTP request latency estimate for its server. An advantage of such a mechanism is that it does not generate any extra requests. However, the probabilistic selection leads also to performance degradation as shown in [35] because some requests are satisfied from servers that are known to be sub-optimal. We, therefore, chose a different approach where the client applet refreshes its latency information for each server with the most recent sample that is older than *time-to-live* (TTL) minutes. The refreshment is done by sending an *asynchronous* `HEAD` request to the server. Therefore, the latency estimate refreshment does not impact the response time perceived by the user. On the other hand, the asynchronous samples lead to an extra network traffic. However, the volume of such traffic can be explicitly controlled by setting the parameter TTL.
Upon sending a request to a server, the client applet sets a timeout. If the timeout expires, the applet considers the original server as failed and selects the resource on the best server among the remaining servers that replicate the resource. The timeout should reflect the response time of the server. For example, a server located overseas should have a higher timeout than a server located within the same WAN. We chose to set the timeout to a T-percentile of request latency in order to reuse the statistical information collected for other purposes. T is a system parameter and typically should be set relatively high (e.g. 99) in order to base the timeout on a pessimistic estimate.
After the timeout for a request sent to a server expires, the applet marks the server entry in the latency table as "failed". For every "failed" server, the applet keeps polling the server by *asynchronously* sending a HEAD request for a randomly chosen resource every F seconds until a response is received. Initially, F is set to a default value that can be for example the mean time-to-repair (MTTR) of Internet servers measured in [29]. Subsequently, the value of F is doubled each time an asynchronous probe is sent to the server. After receiving a response, the value of F is reset to its default value. The default value of F is a system parameter. The pseudo-code of the replica selection algorithm can be found in Figure 4.

4.2 Experimental Evaluation

We compared the efficiency of the original Refresh algorithm with several other algorithms used for HTTP re-

```
Input: $d$ - requested resource
        $R$ - available servers replicating resource $d$
        $L$ - latency table
Output: $s$ - server selected to satisfy request on $d$
while ($R$ nonempty) do
  if (all servers in $R$ have expired entries in $L$) then
    $s$ := randomly selected server from $R$;
  else
    $s$ := server from $R$ with minimal $S$-percentile of latency;
  fi
  send a request to server $s$;
  $timeout$ := $T$-percentile of latency for $s$;
  if ($timeout$ expires) then
    mark entry of server $s$ in $L$ as "failed";
    remove server $s$ from $R$;
    send asynchronous request to $s$ after $F$ seconds until $s$ responds
      and double $F$ each time a request is sent;
    if (response received) then
      mark entry of server $s$ in $L$ as "available";
      include server $s$ in $R$ if no response received;
      reset $F$ to its default value;
    fi
  fi
  if (response received) then
    update estimates of latency average and variance in $L$ for server $s$;
  fi
  if (any server in $R$ has expired entry in $L$) then
    $s'$ := server with the oldest expired entry in $L$;
    send asynchronous request to $s'$;
    depending on response either update $L$ or mark as "failed";
  fi
od
```

Figure 4: **Pseudo-code of replica selection algorithm.**

source replica selection in [35]. We simulated replicated resources by measuring the latencies of HTTP requests sent for resources residing on 5 or 50 most popular servers in a client trace we collected at Northwestern University. In summary, we found that the Refresh algorithm improved the HTTP request latency compared with the other algorithms described in the literature on average by 55%. The Refresh algorithm improved latency on average by 69% compared with a system using only a single server (i.e. no resource replication). More details on the experimental evaluation can be found in [35].

Rather than repeating the experiments from [35], we concentrate here on the accuracy of the estimates used by the above described extension of the Refresh algorithm. The experiments also reveal surprising characteristics of the behavior of HTTP request latency[4]. In the first experiment, we compare the accuracy of HTTP request latency prediction based on an average calculated using the recursive formula (2). To collect the performance data for the comparison, we measured the HTTP request latencies of the fifty most popular servers outside of Northwestern University campus that were referenced in client traces from [35]. Each server was polled with a 1 minute period[5] from a single client at Northwestern University for a period of approximately three days. All together, we collected 228,194 samples of HTTP request latencies. At each step of the experiment, we estimated the next latency sample using the recursive formula (2). The estimate depends on a factor r that determines the weight given to the most recent sample. Figure 5 shows the mean of relative prediction error for various values of r. First, the results show that the HTTP request latency can be predicted relatively accurately from its past samples. For example, even when the sampling interval is increased from 1 minute to 10 and 100 minutes, the mean of relative prediction error is relatively low as shown in Figure 5. Second, the experiment also shows that the smaller the weight given to older samples, the better the accuracy of prediction. Such a behavior can be partly explained by existence of "peaks" on the HTTP request latency curve. The larger the value of r, the faster can the average estimate "forget" the value of the "peak". However, even after filtering out the peaks (all values 5 or 3 times the magnitude of average), we still observed qualitatively similar behavior to that shown in Figure 5. Consequently, we *conjecture* that the "memoryless" behavior is an intrinsic property of the distribution of HTTP request latency (and response time). Finally, we also verified that the accuracy of HTTP request latency prediction based on the recursive formula (5) is as good as the accuracy of prediction based on sliding window used in [35] that lead to a higher storage space overhead.

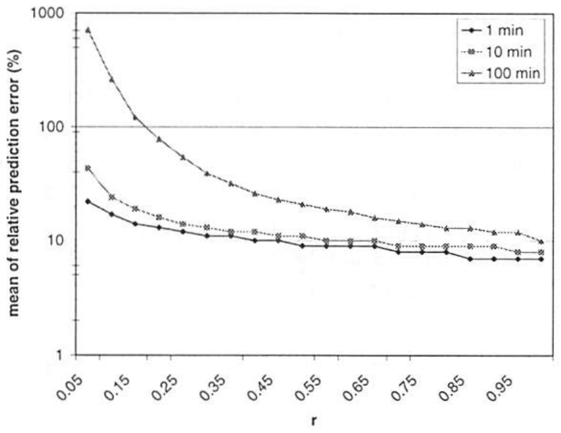


Figure 5: **Latency prediction based on recursive formula.**

The feasibility of our approach for latency estimate refreshment depends on the stability of HTTP request latency. If the latency is unstable, then a small value of TTL must be selected to keep the estimates reasonably close to their current values. Consequently, a large num-

[4]In all experiments we measured also HTTP response time and found its behavior fairly close to that of HTTP request latency.

[5]We did not use a higher polling rate as it could be interpreted as a denial-of-service attack.

ber of extra requests is sent only to keep the latency table up-to-date. Therefore, we used the experimental data described above to evaluate the stability of HTTP request latency and gain an insight to selection of the TTL parameter. For each HTTP request latency sample s_0, we define a (p,q)-*stable period* as the maximal number of samples immediately following s_0 such that at least $p\%$ of samples are within a relative error of $q\%$ of the value of s_0 (this property must hold also for all prefixes of the interval). The stability of a HTTP request latency series can be characterized by the mean length of (p,q)-stable periods over all the samples. Figure 6 shows the means of length of (p,q)-stable periods for various settings of parameters p and q. The results indicate that the HTTP request latency is relative stable. For example, the mean length of the $(90,10)$-stable period is 41 minutes and the mean length of the $(90,30)$-stable period is 483 minutes.

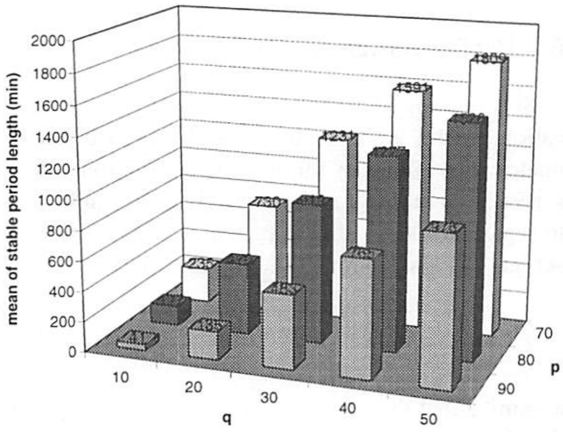


Figure 6: **Latency stability.**

5 Web++ Server Design

The Web++ server is responsible for

- Pre-processing of resources sent to the client.
- Creation and destruction of resource replicas.
- Maintaining consistency of the Replication Directory and replicated resources.
- Tracking the server load.

The server functionality is implemented by extending an existing Web server with a Java servlet. The current implementation of Web++ servlet consists of approximately 24 KB of Java bytecode. The Web++ servlet can be configured as either *server* or *proxy*. In the server configuration, the requested resources are satisfied locally either from disk or by invoking another servlet or CGI script. In the proxy configuration, each request is forwarded as a HTTP request to another server. The server configuration can be used if the server to be extended supports servlet API. If the existing server does not support servlet API, it can be still extended with a server-side proxy server that supports the servlet API (such as the W3C Jigsaw server which is freely available).

The viability of the Web++ design depends on the overhead of Web++ servlet invocation. In Figure 7 we compare the average service time of direct access to static resources and access via Web++ servlet. The servlet access includes also the overhead of resource pre-processing. The client and server executed on the same machine[6] to keep the impact of network overhead minimal. For the purpose of the experiment, we disabled caching of pre-processed resources described in Section 5.1. We found that the servlet-based access to resources leads to a 13.6% service time increase on average, and 5% and 17.6% increase in the best and worst cases. On average, the increase in service time is 3.9 ms, which is more than two orders of magnitude smaller that the response time for access to resources over the Internet (see Figures 11 and 13). We therefore conclude that even with no caching the Web++ servlet overhead is minimal.

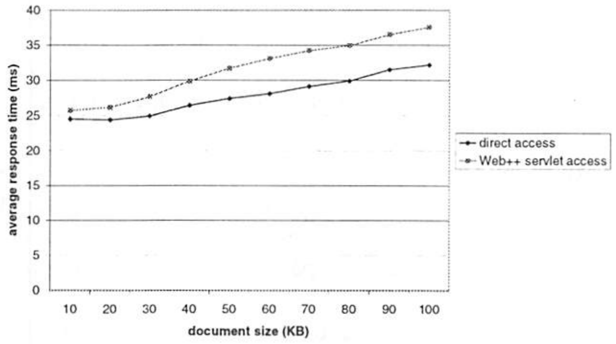


Figure 7: **Web++ servlet overhead.**

5.1 Resource Pre-processing

After receiving an HTTP GET request, the servlet first obtains the resource corresponding to the requested URL. The exact method of obtaining the resource depends on whether the resource is static or dynamic and whether the servlet is configured as a proxy or a server. If the resource is a HTML text, the Web++ servlet pre-processes the resource by including a reference to the Web++ client applet and expanding references to all logical URLs with JavaScript event handlers as described in Section 3. The logical URLs are found by a regular expression match for

[6]Sun SparcStation with 128 MB of RAM running Solaris 2.5 and Java Web Server 1.1.

URL strings following the <HREF> and <SRC> HTML tags. The matched strings are then compared against the local replication directory. If a match is found, the server emits the modified text into the HTML resource, otherwise the matched URL is left unmodified.
To amortize the post-processing overhead, the Web++ servlet caches the post-processed resources in its main memory. Caching of dynamically generated resources is a complex issue studied elsewhere [25] and its full exposition exceeds the scope of this paper. In the current implementation of the Web++ servlet, we limit cache consistency enforcement to testing of the `Last-Modified` HTTP header of the cached resource and the most recent modification UNIX timestamp for static resources corresponding to local files.
Web++ servlets exchange entries of their replication directories in order to inform other servers about newly created or destroyed resource replicas [41].

5.2 Resource replica management

The Web++ servlet provides a support for resource replica creation, deletion and update. The replica management actions are carried on top of HTTP `POST`, `DELETE` and `PUT` operations with the formats shown in Figure 8.

creation:	`POST`	logical URL	<*resource*>
deletion:	`DELETE`	logical URL	
update:	`PUT`	logical URL	<*resource*>

Figure 8: **Format of replication operations.**

After receiving `POST`, `DELETE` or `PUT` requests, the servlet creates a new copy of the resource, deletes the resource or updates the resource specified by the logical URL depending on the type of operation. In addition, if the operation is either `POST` or `DELETE`, the servlet also updates its local replication directory to reflect either creation or destruction of its replica. The servlet also propagates the update to other servers using the algorithm described in [41] that guarantees eventual consistency of the replication directories. We assume that such an exchange will occur only among the servers within the same administrative domain (for scalability and security reasons). Servers in separate administrative domains can still help each other to resolve the logical URL references by exporting a name service that can be queried by servers outside of the local administrative domain. The name service can be queried by sending an HTTP `GET` request to a well known URL. The logical URL to be resolved is passed as an argument in the URL.
The Web++ servlet provides the basic operations for creation, destruction and update of replicated resource. Such operations can be used as basic building blocks for algorithms that decide if a new replica of a resource should be created, on which server it should be created or how the replicas should be kept consistent in the presence of updates. Web++ provides a framework within which such algorithms can be implemented. In particular, each of the servlet handlers for `POST`, `DELETE` and `PUT` operations can invoke user-supplied methods (termed *pre-filters* and *post-filters*) either before or after the handler is executed. Any of the operations mentioned above can be implemented as a pre or post filter. Algorithms that dynamically decide whether the system should be expanded with an additional server have been described in [9, 42]. Algorithms that dynamically determine near optimal placement of replicated resources within a network have been studied in [5, 43]. Finally, algorithms for replica consistency maintenance have been described in [24, 40, 8]. However, in order to apply them to the Web these algorithms need to be extended with new performance metrics as well as take into account the hierarchical structure of the Internet. Study of such algorithms exceeds the scope of this paper.

6 Performance Analysis

In Section 1 we identified two main reasons for data replication on the Web: better reliability and better performance. The reliability improvement is obvious: If a single server has a mean time to failure $MTTF_1$ and mean time to repair $MTTR_1$, then a system with n fully replicated servers has a mean time to failure given by

$$MTTF_n \approx \frac{MTTF_1^n}{n \cdot MTTR_1^{n-1}}$$

assuming that the server failures are statistically independent [21]. Clearly, the mean time to failure improves with the number of replica servers. In order to ascertain the performance gains of resource replication, we conducted a live experiment with the Web++ system.

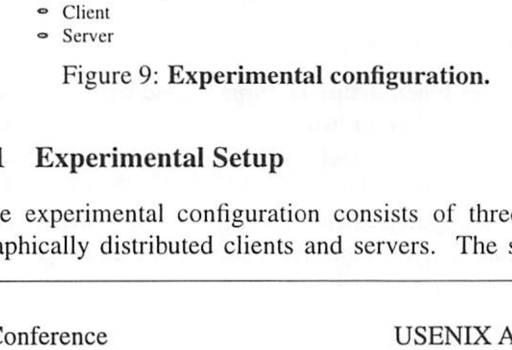


Figure 9: **Experimental configuration.**

6.1 Experimental Setup

The experimental configuration consists of three geographically distributed clients and servers. The servers were located at Stanford University (Palo Alto, US west coast), University of Kentucky (Lexington, US east coast) and University of Saarbruecken (Saarbruecken, Germany). On each site we installed a Web++ server consisting of Sun's Java Web Server 1.1 extended with the Web++ servlet. The clients were located at University of California at Santa Barbara (Santa Barbara, US west coast), Bell Labs (Murray Hill, US east coast) and Charles University (Prague, Czech Republic). For the purpose of the experiment, we converted the Web++ client applet into an application and ran it under the JDK 1.0 Java interpreter [7]. The parameter settings of our experimental configuration are shown in Figure 9.

parameter	value (unit)
r	0.95
TTL	41 (min)
default F	28.8 (hours)
S	55
T	99.9

Figure 10: **Parameter settings for Web++ system.**

The workload on each client consisted of 500 GET requests for resources of a fixed size. We considered 0.5KB, 5KB, 50KB and 500KB files fully replicated on all three servers. The advantage of such a workload is that it allows us to study the benefits of replication in isolation for each resource size. It is also relatively straightforward to estimate the performance gains for a given workload mix (e.g. SPECweb96 [2]). Each client generated a new request only after receiving a response to a previously sent request. We executed the entire workload both during peak office hours (noon EST) and during evening hours (6pm EST). In each experiment we report the mean of response time measured across all requests sent by all clients.

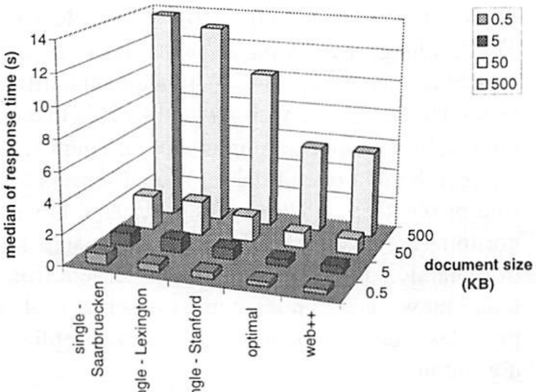


Figure 11: **Absolute response time - noon.**

We compared the performance of our Web++ system with a system that uses only a *single* server (i.e. no resource replication) and an *optimal* system that sends three requests in parallel and waits for the first response. The parameters of our Web++ system (shown in Figure 10) were set based on the sensitivity study conducted in Section 4. In particular we set parameter r to 0.95 since values close to 1 lead to best prediction accuracy as shown in Figure 5. Time-to-live (TTL) was set to 41 minutes that corresponds to the average length of a (90,10)-stable periods as shown in Figure 6. The default value of F was set to 28.8 hours that corresponds to the mean time-to-repair for Internet servers as measured in [29]. We set the percentile S to 55 to keep the impact of the variance on server selection minimal (the purpose of variance is to only break ties for servers with similar average latency). Finally, we set the percentile T to 99.9 to minimize the number of false timeouts [8].

In contrast to the experiments in [35], in the experiments reported here we tested a complete system (Web++ clients and servers). Since we had a control over the server side, we were able to compare the HTTP request response times for resources of different pre-determined sizes. Finally, we also used three geographically distributed clients as opposed to a single client in [35]. On the other hand, all resources were replicated only on three servers (as opposed to five and fifty in [35]). This limitation was imposed on us by the number of accounts we could obtain for the purpose of the experiment.

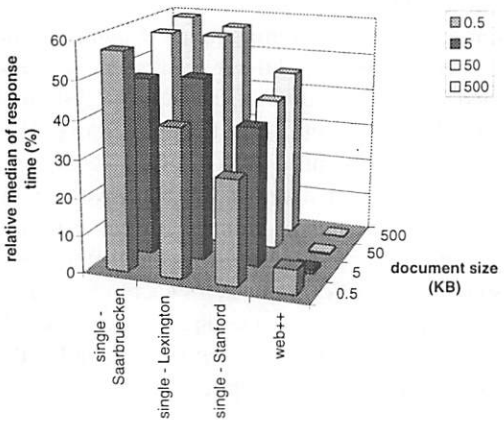


Figure 12: **Relative response time - noon.**

6.2 Experimental Results

We found that Web++ improves the response time during the peak hours on the average by 47.8%, at least by 27.8% and at most by 59.1% when compared to the single any single server system. At the same time, it degrades the response time relative to the optimal system on average by 2.2%, at least by 0.2% and at most by 7.1%. Not

[7]Some of the clients ran on platforms that did not support JDK1.1 at the time of experiment.

[8]In most cases the calculated timeout was larger than the timeout of the underlying `java.net.URLConnection` implementation

surprisingly, we found that the performance benefits of Web++ are weaker during the evening hours. In particular, we found that Web++ improves the response time on the average by 25.5%, at most by 58.9% and in the worst case it may degrade performance by 1.4%. We also found that Web++ degrades the response time with respect to the optimal system on average by 25.5%, at least by 7.8% and at most by 31%. Throughout all the experiments we found that Web++ did not send more than 6 extra requests to refresh its latency table (compared with three times as many requests sent by the optimal system!).

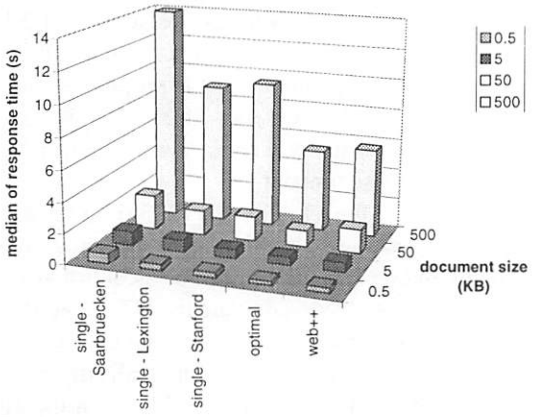


Figure 13: **Absolute response time - evening.**

The experimental results can be found in Figures 11 - 14. Figures 11 and 13 show the median of the response time during office and evening hours. Figures 12 and 14 show the relative median of the response time with respect to the median response time of the optimal system. Compared with the experimental results reported in [35] (an average 69% improvement in HTTP request *latency*), the results reported here (an average 37% improvement in HTTP request *response time*) indicate a weaker improvement in performance. We believe that the difference is due to a smaller number of replicas for each resource in the experiments reported here (3 compared with 5 and 50 in [35]). The bigger the number of replicas the higher the probability that a client finds a replica "close" to it.

7 Related Work

The use of replication to improve system performance and reliability is not new. For example, process groups have been successfully incorporated into the design of some transaction monitors [21]. The performance benefits of Web server replication were first observed in [7, 23]. The authors also pointed out that resource replication may eliminate the consistency problems introduced by proxy server caching.

The architecture of the Web++ client is closely related to Smart Clients [44]. In fact, the Web++ client is a specific instance of a Smart Client. While Yoshikawa et. al.

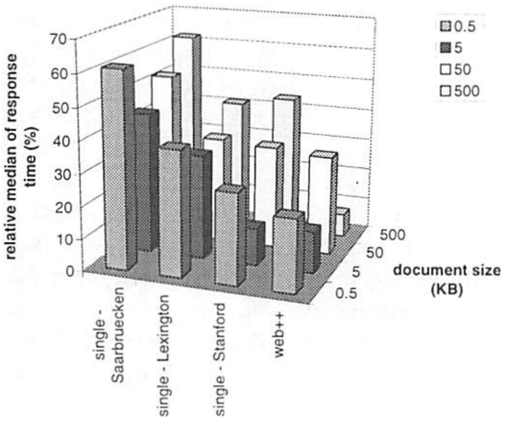


Figure 14: **Relative response time - evening.**

describe smart clients implementing FTP, TELNET and chat services, we concentrate on the HTTP service. We provide a detailed description how the client applet can be integrated with the browser environment. In addition, we describe a specific algorithm for selection of replicated HTTP servers and provide its detailed performance analysis. Finally, we describe the design and implementation of the server end.

Cisco DistributedDirector is a product that provides DNS-level replication [14]. DistributedDirector requires full server replication, because the redirection to a specific server is done at the network level. As argued in the Introduction, this may be impractical for several reasons. The DNS-level replication also leads to several problems with recursive DNS queries and DNS entry caching on the client. DistributedDirector relies on a modified DNS server that queries server-side agent to resolve a given hostname to an IP address of a server which is closest to the querying DNS client[9]. DistributedDirector supports several metrics including various modification of routing distance (#hops), random selection and round-trip-time (RTT). However, it is not clear from [14] how the RTT delay is measured and how it is used to select a server.

The Caching goes Replication (CgR) is a prototype of replicated web service [4]. A fundamental difference between the designs of Web++ and CgR is that CgR relies on a client-side proxy to intercept all client requests and redirect them to one of the replicated servers. The client-side proxy keeps track of all the servers, however no algorithms are given to maintain the client state in presence of dynamic addition or removal of servers from the system. Our work does not assume full server replication and provides a detailed analysis of resource replica selection algorithm.

Proxy server caching is similar to server replication in that it also aims at minimizing the response time by placing resources "nearby" the client [20, 30, 3, 10, 28, 36, 15, 37,

[9]The DNS client may be in a completely different location than the Web client if a recursive DNS query is used

13]. The fundamental difference between proxy caches and replicated Web servers is that the replicated servers know about each other. Consequently, the servers can enforce any type of resource consistency unlike the proxy caches, which must rely on the expiration-based consistency supported by the HTTP protocol. Secondly, since the replicated servers are known to content providers, they provide an opportunity for replication of an active content. Finally, the efficiency of replicated servers does not depend on access locality, which is typically low for Web clients (most client trace studies show hit rates below 50% [3, 10, 15, 28, 36]).

Several algorithms for replicated resource selection have been studied in [11, 22, 23, 38, 17, 35, 27, 19]. A detailed discussion of the subject can be found in Section 4.

8 Conclusions

Web++ is a system that aims at improving the response time and the reliability of Web service. The improvement is achieved by geographic replication of Web resources. Clients reduce their response time by satisfying requests from "nearby" servers and improve their reliability by failing over to one of the remaining servers if the "closest" server is not available. Unlike the replication currently deployed by most Web sites, the Web++ replication is completely transparent to the browser user.

As the major achievement of our work, we demonstrate that it is possible to provide user-transparent Web resource replication without any modification on the client-side and using the existing Web infrastructure. Consequently, such a system for Web resource replication can be deployed relatively quickly and on a large scale. We implemented a Web++ applet that runs within Microsoft Explorer 4.x and Netscape Navigator 4.x browsers. The Web++ servlet runs within Sun Java Web Server. We currently explore an implementation of Web++ client based on Microsoft ActiveX technology.

We demonstrated the efficiency of the entire system on a live Internet experiment on six geographically distributed clients and servers. The experimental results indicate that Web++ improves the response time on average by 47.8% during peak hours and 25.5% during night hours, when compared to the performance of a single server system. At the same time, Web++ generates only a small extra message overhead that did not exceed 2% in our experiments.

Web++ provides a framework for implementing various algorithms that decide how many replicas should be created, on which servers the replicas should be placed and how the replicas should be kept consistent. We believe that all of these issues are extremely important and are a subject of our future work.

9 Acknowledgments

We would like to thank Reinhard Klemm for numerous extremely helpful discussions on the subject, sharing his performance results with us and proving us with a code of WebCompanion. Our understanding of the subject was also helped by discussions with Yair Bartal. Finally, our thanks belong to Amr El Abbadi, Divy Agrawal, Tomáš Doležal, Hector Garcia-Molina, Jim Griffioen, Jaroslav Pokorný, Markus Sinnwell and Gerhard Weikum who helped us with opening accounts on their systems, installing necessary software and keeping the systems running. Without their help, the study of Web++ performance would have been impossible.

References

[1] Internet weather report (IWR). available at `http://www.mids.org`.

[2] The workload for the SPECweb96 benchmark. available at `http://ftp.specbench.org/osg/web96/workload.html`, 1998.

[3] M. Abrams, C. Standridge, G. Abdulla, S. Williams, and E. Fox. Caching proxies: Limitatons and potentials. *Computer Networks and ISDN Systems*, 28, 1996.

[4] M. Baentsch, G. Molter, and P. Sturm. Introducing application-level replication and naming into today's web. *Computer Networks and ISDN Systems*, 28, 1996.

[5] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proceeding of the 24th Annual ACM Symposium on Theory and Computing*, 1992.

[6] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URL). IETF Network Working Group, RFC 1738, 1994.

[7] A. Bestavros. Demand-based reource allocation to reduce traffic and balance load in distributed information systems. In *Proceeding of the 7th IEEE Symposium on Parallel and Distributed Processing*, 1995.

[8] Y. Breitbart and H. F. Korth. Replication and consistency: Being lazy helps sometimes. In *Proceeding of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1997.

[9] Y. Breitbart, R. Vingralek, and G. Weikum. Load control in scalable distributed file structures. *Distributed and Parallel Databases*, 4(4), 1996.

[10] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[11] M. Crovella and R. Carter. Dynamic server selection in the internet. In *Proceeding of IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, 1995.

[12] O. Damani, P. Chung, Y. Huang, C. Kintala, and Y. Wang. ONE-IP: techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems*, 29, 1997.

[13] P. Danzig and K. Swartz. Transparent, scalable, fail-safe web caching. Technical Report TR-3033, Network Appliance, 1998.

[14] K. Delgadillo. Cisco DistributedDirector. available at http://www.cisco.com/warp/public/751/distdir/dd_wp.html, 1998.

[15] B. Duska, D. Marwood, and M. Feeley. The measured access characteristics of world-wide-web client proxy caches. In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[16] R. Farrell. Review: Distributing the web load. *Network World*, 1997.

[17] Z. Fei, S. Bhattacharjee, E. Zegura, and M. Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proceeding of IEEE INFOCOM'98*, 1998.

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol - HTTP/1.1. IETF Network Working Group, RFC 2068, 1997.

[19] P. Francis. A call for an internet-wide host proximity service (HOPS). available at `http://www.ingrid.org/hops/wp.html`.

[20] S. Glassman. A caching relay for the world wide web. *Computer Networks and ISDN Systems*, 27, 1994.

[21] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[22] J. Guyton and M. Schwartz. Locating nearby copies of replicated internet servers. In *Proceeding of ACM SIGCOMM'95*, 1995.

[23] J. Gwertzman and M. Seltzer. The case for geographical push cashing. In *Proceeding of the 5th Workshop on Hot ZTopic in Operating Systems*, 1995.

[24] A. Helal, A. Heddaya, and B. Bhargava. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, 1986.

[25] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[26] E. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27, 1994.

[27] R. Klemm. WebCompanion: A friendly client-side web prefetching agent. *IEEE Transactions on Knowledge and Data Engineering*, 1999.

[28] T. Kroeger, D. Long, and J. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[29] D. Long, A. Muir, and R. Golding. A longitudinal survey of internet host reliability. In *Proceeding of the 14th Symposium on Reliable Distributed Systems*, 1995.

[30] A. Luotonen and K. Altis. World-wide web proxies. *Computer Networks and ISDN Systems*, 27, 1994.

[31] J. MacGregor and T. Harris. The exponentially weighted moving variance. *Journal of Quality Technology*, 25(2), 1993.

[32] S. Manley and M. Seltzer. Web facts and fantasy. In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*, 1998.

[33] M. Mitzenmacher. How useful is old information? Technical Report TR-1998-002, Systems Research Center, Digital Equipment Corporation, 1998.

[34] H. Frystyk Nielsen, J. Gettys, A. Baird-Smith, Eric Prud'hommeaux, H. Wium Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceeding of ACM SIGCOMM'97*, 1997.

[35] M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek. Selection algorithms for replicated web servers. In *Proceeding of the Workshop on Internet Server Performance*, 1998. available at `http://lilac.ece.nwu.edu:1024/publications/WISP98/final/SelectWeb1.html`.

[36] P. Scheuermann, J. Shim, and R. Vingralek. A case for delay-conscious caching of web documents. *Computer Networks and ISDN Systems*, 29, 1997.

[37] P. Scheuermann, J. Shim, and R. Vingralek. An unified algorithm for cache replacement and consistency in web proxy servers. In *Proceeding of the Workshop on Data Bases and Web*, 1998.

[38] S. Seshan, M. Stemm, and R. Katz. SPAND: shared passive network performance discovery. In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[39] K. Sollins. Architectural principles for uniform resource name resolution. IETF Network Working Group, RFC 2276, 1995.

[40] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceeding of the ACM SIGOPS Symposium on Principles of Operating Systems*, 1995.

[41] R. Vingralek, Y. Breitbart, M. Sayal, and P. Scheuermann. Architecture, design and analysis of web++. Technical report, CPDC-TR-9902-001, Northwestern University, Department of Electrical and Computer Engineering, 1999. available at `http://www.ece.nwu.edu/cpdc/HTML/techreports.html`.

[42] R. Vingralek, Y. Breitbart, and G. Weikum. SNOWBALL: Scalable storage on networks of workstations. *Distributed and Parallel Databases*, 6(2), 1998.

[43] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2), 1997.

[44] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using smart clients to build scalable services. In *Proceedings of USENIX'97*, 1997.

Efficient Support for P-HTTP in Cluster-Based Web Servers

Mohit Aron Peter Druschel Willy Zwaenepoel
Department of Computer Science
Rice University

Abstract

This paper studies mechanisms and policies for supporting HTTP/1.1 persistent connections in cluster-based Web servers that employ content-based request distribution. We present two mechanisms for the efficient, content-based distribution of HTTP/1.1 requests among the back-end nodes of a cluster server. A trace-driven simulation shows that these mechanisms, combined with an extension of the locality-aware request distribution (LARD) policy, are effective in yielding scalable performance for HTTP/1.1 requests. We implemented the simpler of these two mechanisms, *back-end forwarding*. Measurements of this mechanism in connection with extended LARD on a prototype cluster, driven with traces from actual Web servers, confirm the simulation results. The throughput of the prototype is up to four times better than that achieved by conventional weighted round-robin request distribution. In addition, throughput with persistent connections is up to 26% better than without.

1 Introduction

Clusters of commodity workstations are becoming an increasingly popular hardware platform for cost-effective high performance network servers. Achieving scalable server performance on these platforms is critical to delivering high performance to users in a cost-effective manner.

State-of-the-art cluster-based Web servers employ a front-end node that is responsible for distributing incoming requests to the back-end nodes in a manner that is transparent to clients. Typically, the front-end distributes the requests such that the load among the back-end nodes is balanced. With *content-based request distribution*, the front-end additionally takes into account the content or type of service requested when deciding to which back-end node a client request should be assigned.

Content-based request distribution allows the integration of server nodes that are specialized for certain types of content or services (e.g., audio/video), it permits the partitioning of the server's database for scalability, and it enables clever request distribution policies that improve performance. In previous work, we proposed *locality-aware request distribution* (LARD), a content-based policy that achieves good cache hit rates in addition to load balance by dynamically partitioning the server's working set among the back-end nodes [23].

In this paper, we investigate mechanisms and policies for content-based request distribution in the presence of HTTP/1.1 [11] persistent (keep-alive) client connections (P-HTTP). Persistent connections allow HTTP clients to submit multiple requests to a given server using a single TCP connection, thereby reducing client latency and server overhead [19]. Unfortunately, persistent connections pose problems for clusters that use content-based request distribution, since requests in a single connection may have to be assigned to different back-end nodes to satisfy the distribution policy.

This paper describes efficient mechanisms for content-based request distribution and an extension of the LARD policy in the presence of HTTP/1.1 connections. It presents a simulation study of these mechanisms, and it reports experimental results from a prototype cluster implementation. The results show that persistent connections can be supported efficiently on cluster-based Web servers with content-based request distribution. In particular, we demonstrate that using *back-end forwarding*, an extended LARD policy achieves up to 26% better performance with persistent connections than without.

The rest of the paper is organized as follows. Section 2 provides some background information on HTTP/1.1 and LARD, and states the problems posed by HTTP/1.1 for clusters with content-based request distribution. Section 3 considers mechanisms for achieving content-based request distribution in the presence of HTTP/1.1 persistent connections. The extended LARD policy is presented in Section 4. Section 5 presents a performance analysis of our request distribution mechanisms. A simulation study of the various mechanisms and the extended LARD policy is described in Section 6. Section 7 discusses a prototype implementation, and

Section 8 reports measurement results obtained using that prototype. We discuss related work in Section 9, and conclude in Section 10.

2 Background

This section provides background information on persistent connections in HTTP/1.1, content-based request distribution, and the LARD strategy. Finally, we state the problem that persistent connections pose to content-based request distribution.

2.1 HTTP/1.1 persistent connections

Obtaining an HTML document typically involves several HTTP requests to the Web server, to fetch embedded images, etc. Browsers using HTTP/1.0 [5] send each request on a separate TCP connection. This increases the latency perceived by the client, the number of network packets, and the resource requirements on the server [19, 22].

HTTP/1.1 enables browsers to send several HTTP requests to the server on a single TCP connection. In anticipation of receiving further requests, the server keeps the connection open for a configurable interval (typically 15 seconds) after receiving a request . This method amortizes the overhead of establishing a TCP connection (CPU, network packets) over multiple HTTP requests, and it allows for pipelining of requests [19]. Moreover, sending multiple server responses on a single TCP connection in short succession avoids multiple TCP slow-starts [29], thus increasing network utilization and effective bandwidth perceived by the client.

RFC 2068 [11] specifies that for the purpose of backward compatibility, clients and servers using HTTP/1.0 can use persistent connections through an explicit HTTP header. However, for the rest of this paper, HTTP/1.0 connections are assumed not to support persistence. Moreover, this paper does not consider any new features in HTTP/1.1 over HTTP/1.0 other than support for persistent connections and request pipelining.

2.2 Content-based Request Distribution

Content-based request distribution is a technique employed in cluster-based network servers, where the front-end takes into account the service/content requested when deciding which back-end node should serve a given request. In contrast, the purely load-based schemes like *weighted round-robin* (WRR) used in commercial high performance cluster servers [15, 8] distribute incoming requests in a round-robin fashion, weighted by some measure of load on the different back-end nodes.

The potential advantages of content-based request distribution are: (1) increased performance due to improved hit rates in the back-end's main memory caches, (2) increased secondary storage scalability due to the ability to partition the server's database over the different back-end nodes, and (3) the ability to employ back-end nodes that are specialized for certain types of requests (e.g., audio and video).

With content-based request distribution, the front-end must establish the TCP connection with the client *prior* to assigning the connection to a back-end node, since the nature and the target[1] of the client's request influences the assignment. Thus, a mechanism is required that allows a chosen back-end node to serve a request on the TCP connection established by the front-end. For reasons of performance, security, and interoperability, it is desirable that this mechanism be transparent to the client. We will discuss mechanisms for this purpose in Section 3.

2.3 Locality-aware request distribution

Locality-aware request distribution (LARD) is a specific strategy for content-based request distribution that focuses on the first of the advantages cited above, namely improved cache hit rates in the back-ends [23]. LARD strives to improve cluster performance by *simultaneously* achieving load balancing and high cache hit rates at the back-ends.

Figure 1 illustrates the principle of LARD in a cluster with two back-ends and a working set of three targets (A, B, and C) in the incoming request stream. The front-end directs all requests for A to back-end 1, and all requests for B and C to back-end 2. By doing so, there is an increased likelihood that the request finds the requested target in the cache at the back-end.

In contrast, with a round-robin distribution of incoming requests, requests for all three targets will arrive at both back-ends. This increases the likelihood of a cache miss, if the sum of the sizes of the three targets, or, more generally, if the size of the working set exceeds the size of the main memory cache at an individual back-end node. Thus, with a *round-robin* distribution, the cluster does not scale well to larger working sets, as *each* node's main memory cache has to fit the entire working set. With LARD, the effective cache size approaches the *sum* of the individual node cache sizes. Thus, adding nodes to a cluster can accommodate both increased traffic (due to additional CPU power) and larger working sets (due to the increased effective cache size).

[1] In the following discussion, the term *target* is used to refer to a Web document, specified by a URL and any applicable arguments to the HTTP GET command.

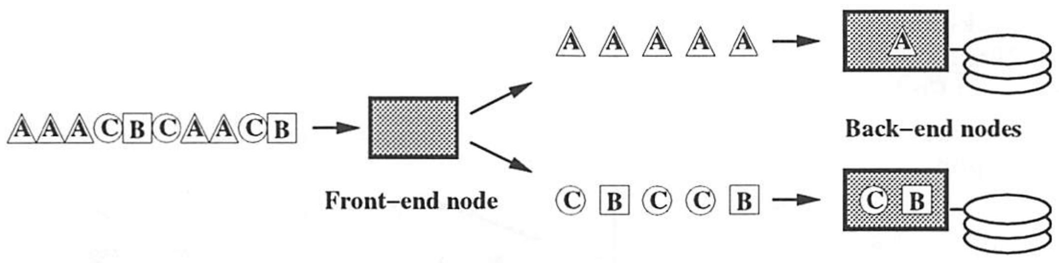


Figure 1: Locality-Aware Request Distribution

2.4 The problem with HTTP/1.1

HTTP/1.1 persistent connections pose a problem for clusters that employ content-based request distribution, including LARD. The problem is that existing, scalable mechanisms for content-based distribution operate at the granularity of TCP connections. With HTTP/1.1, multiple HTTP requests may arrive on a single TCP connection. Therefore, a mechanism that distributes load at the granularity of a TCP connection constrains the feasible distribution policies, because all requests arriving on a given connection must be served by a single back-end node.

This constraint is most serious in clusters where certain requests can only be served by a subset of the back-end nodes. Here, the problem is one of correctness, since a back-end node may receive requests that it cannot serve.

In clusters where each node is capable of serving any valid request, but the LARD policy is used to partition the working set, performance loss may result since a back-end node may receive requests not in its current share of the working set. As we will show in Section 6, this performance loss can more than offset the performance advantages of using persistent connections in cluster servers.

3 Mechanisms for content-based request distribution

A front-end that performs content-based request distribution must establish a client HTTP connection before it can decide which back-end node should serve the request. Therefore, it needs a mechanism that allows it to have the chosen back-end node serve request(s) on the established client connection. In this section, we discuss such mechanisms.

The simplest mechanisms work by having the front-end "redirect" the client browser to the chosen back-end node, by sending an HTTP redirect response, or by returning a Java applet that contacts the appropriate back-end when executed in the browser [1].

These mechanisms work also for persistent connections, but they have serious drawbacks. The redirection introduces additional delay; the address of individual back-end nodes is exposed to clients, which increases security risks; and, simple or outdated browsers may not support redirection. For these reasons, we only consider client-transparent mechanisms in the remainder of this paper.

3.1 Relaying front-end

A simple client-transparent mechanism is a *relaying front-end.* Figure 2 depicts this mechanism and the other mechanisms discussed in the rest of this section. Here, the front-end maintains persistent connections (back-end connections) with all of the back-end nodes. When a request arrives on a client connection, the front-end assigns the request, and forwards the client's HTTP request message on the appropriate back-end connection. When the response arrives from the back-end node, the front-end forwards the data on the client connection, buffering the data if necessary.

The principal advantage of this approach is its simplicity, its transparency to both clients and back-end nodes, and the fact that it allows content-based distribution at the granularity of individual requests, even in the presence of HTTP/1.1 persistent connections.

A serious disadvantage, however, is the fact that all response data must be forwarded by the front-end. This may render the front-end a bottleneck, unless the front-end uses substantially more powerful hardware than the back-ends. It is conceivable that small clusters could be built using as a front-end a specialized layer 4 switch with the ability to relay transport connections. We are, however, not aware of any actual implementations of this approach. Furthermore, results presented in Section 6.1 indicate that, even when the front-end is not a bottleneck, a relaying front-end does not offer significant performance advantages over more scalable mechanisms.

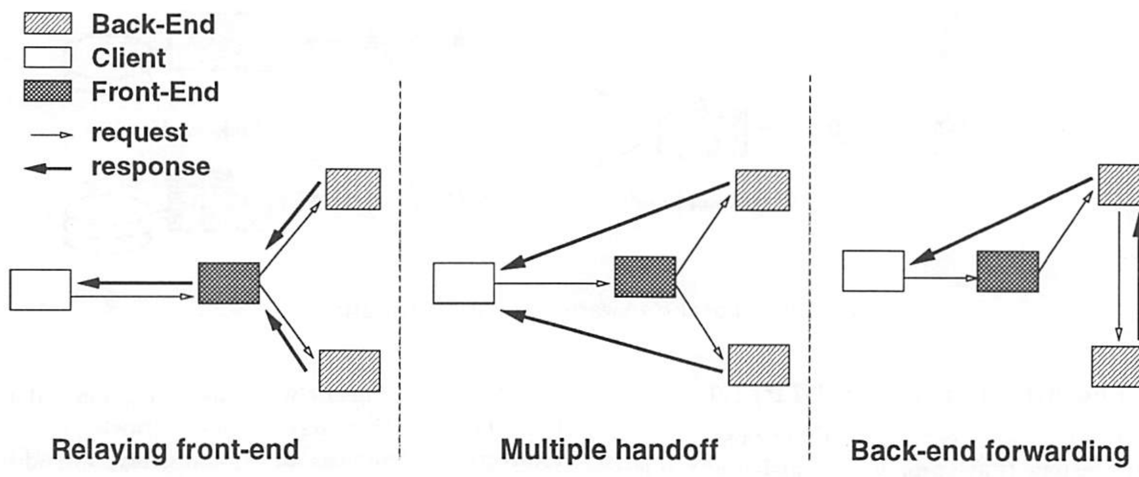


Figure 2: Mechanisms for request distribution

3.2 Multiple TCP connection handoff

A more complex mechanism involves the use of a *TCP handoff protocol* among front-end and back-end nodes. The handoff protocol allows the front-end to transfer its end of an established client connection to a back-end node. Once the state is transferred, the back-end transmits response data directly to the client, bypassing the front-end. Data from the client (primarily TCP ACK packets) are forwarded by the front-end to the appropriate back-end node in an efficient manner.

In previous work, we have designed, implemented, and evaluated a handoff protocol for HTTP/1.0 [23]. This *single handoff* protocol can support persistent connections, but all requests must be served by the back-end node to which the connection was originally handed off.

The design of this handoff protocol can be extended to support HTTP/1.1 by allowing the front-end to migrate a connection between back-end nodes. The advantage of this *multiple handoff* protocol is that it allows content-based request distribution at the granularity of individual requests in the presence of persistent connections. Unlike front-end relaying, the handoff approach is efficient and scalable since response network traffic bypasses the front-end.

The handoff approach requires the operating systems on front-end and back-end nodes to be customized with a vendor-specific loadable kernel module. The design of such a module is relatively complex, especially if multiple handoff is to be supported. To preserve the advantages of persistent connections – reduced server overhead and reduced client latency – the overhead of migrating connections between back-end nodes must be kept low, and the TCP pipeline must be kept from draining during migration.

3.3 Back-end request forwarding

A third mechanism, *back-end request forwarding*, combines the TCP single handoff protocol with forwarding of requests and responses among back-end nodes. In this approach, the front-end hands off client connections to an appropriate back-end node using the TCP single handoff protocol. When a request arrives on a persistent connection that cannot (or should not) be served by the back-end node that is currently handling the connection, the connection is *not* handed off to another back-end node.

Instead, the front-end informs the connection handling back-end node A which other back-end node B should serve the offending request. Back-end node A then requests the content or service in question directly from node B, and forwards the response to the client on its client connection. Depending on the implementation, these "lateral" requests are forwarded through persistent HTTP connections among the back-end nodes, or through a network file system.

The advantages of back-end request forwarding lie in the fact that the complexity and overhead of multiple TCP handoff can be avoided. The disadvantage is the overhead of forwarding responses on the connection handling back-end node. This observation suggests that the back-end request forwarding mechanism is appropriate for requests that result in relatively small amounts of response data. Results presented in Section 6 show that due to the relatively small average content size in today's Web traffic [19, 3], the back-end request forwarding approach is very competitive.

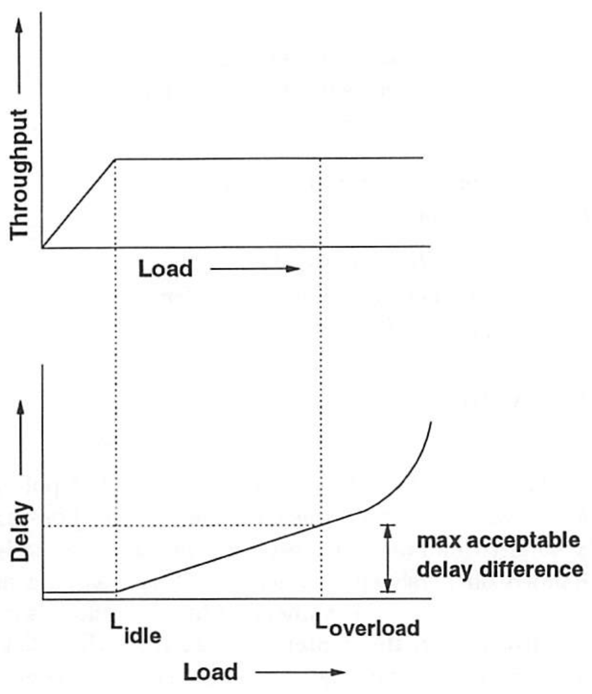


Figure 3: Server Throughput and Delay

4 Policies

This section presents an extension of the LARD policy that works efficiently in the presence of HTTP/1.1 persistent connections, when used with the back-end request forwarding mechanisms presented in the previous section.

Both the front-end relaying mechanism and the TCP multiple handoff mechanism allow requests to be distributed at the granularity of individual requests. As such, they do not place any restriction on the request distribution policies that can be used. In particular, the LARD policy can be used in combination with these mechanisms without loss of locality.

The back-end forwarding mechanism, on the other hand, does place restrictions on the distribution policy, as it mandates that a connection can be handed off to a back-end node only once. If requests arrive on a persistent connection that cannot or should not be served by that back-end node, the policy must instruct that back-end node to forward the request to another back-end node.

We have developed an extension of the LARD policy that can efficiently distribute HTTP/1.1 requests in a cluster that uses the back-end forwarding mechanism. The following subsection briefly presents the standard LARD strategy. Then, we proceed to present our extension.

4.1 The LARD strategy

The LARD strategy yields scalable performance by achieving both load balancing and cache locality at the back-end servers. For the purpose of achieving cache locality, LARD maintains mappings between targets and back-end nodes, such that a target is considered to be cached on its associated back-end nodes. To achieve a balance between load distribution and locality, LARD uses three cost metrics: *cost_balancing*, *cost_locality* and *cost_replacement*. The intuition for the definition of these metrics can be explained using Figure 3, which shows the throughput and delay characteristics of a typical back-end server as a function of load (measured in number of active connections).

The load point L_{idle} defines a value below which a back-end node is potentially underutilized. $L_{overload}$ is defined such that the difference in delay between a back-end node operating at or above this load, compared to a back-end node operating at the point L_{idle}, becomes unacceptable.

The metric *cost_balancing* captures the delay in the servicing of a request because of other queued requests. *Cost_locality*, on the other hand, reflects the delay arising due to the presence or absence of the target in the cache. *Cost_replacement* is a cost that reflects the potential future overhead caused by the replacement of a target in the cache. The three cost metrics are then defined as shown in Figure 4.

The unit of cost (and also of load) is defined to be the delay experienced by a request for a cached target at an otherwise unloaded server. The aggregate cost for sending the request to a particular server is defined as the sum of the values returned by the above three cost metrics. When a request arrives at the front-end, the LARD policy assigns the request to the back-end node that yields the minimum aggregate cost among all nodes, and updates the mappings to reflect that the requested target will be cached at that back-end node[2].

Our experimental results with the Apache 1.3.3 webserver running on FreeBSD-2.2.6 indicate settings of $L_{overload}$ to 130, L_{idle} to 30 and $Miss\ Cost$ to 50. We have used these settings both for our simulator as well as for our prototype results in this paper.

4.2 The extended HTTP/1.1 LARD strategy

The basic LARD strategy bases its choice of a back-end node to serve a given request only on the

[2]Although we present LARD differently than in Pai et al. [23], it can be proven that the strategies are equivalent when $L_{idle} \equiv T_{low}$ and $Miss\ Cost \equiv T_{high} - T_{low}$.

$$cost\_balancing(target, server) = \begin{cases} 0 & Load(server) < L_{idle} \\ Infinity & Load(server) > L_{overload} \\ Load(server) - L_{idle} & otherwise \end{cases}$$

$$cost\_locality(target, server) = \begin{cases} 1 & target\ is\ mapped\ to\ server \\ Miss\ Cost & otherwise \end{cases}$$

$$cost\_replacement(target, server) = \begin{cases} 0 & Load(server) < L_{idle} \\ 0 & target\ is\ mapped\ to\ server \\ Miss\ Cost & otherwise \end{cases}$$

Figure 4: LARD Cost Metrics

current load and the current assignment of content to back-end nodes (i.e., the current partitioning of the working set.) An extended policy that works for HTTP/1.1 connections with the back-end forwarding mechanisms has to consider additional factors, because the choice of a back-end node to serve a request arriving on a persistent connection may already be constrained by the choice of the back-end node to which the connection was handed off. In particular, the policy must make the following considerations:

1. The best choice of a back-end node to handle a persistent connection depends on all the requests expected on the connection.
2. Assigning a request to a back-end node other than the connection handling node causes additional forwarding overhead. This overhead must be weighed against the cost of reading the requested content from the connection handling node's local disk.
3. Given that a requested content has to be fetched from the local disk or requested from another back-end node, should that content be cached on the connection handling node? Caching the content reduces the cost of future requests for the content on the node handling the connection, but it also causes potential replication of the content on multiple back-end nodes, thus reducing the aggregate size of the server cache.

The intuition behind the extended LARD policy is as follows. Regarding (1), due to the structure of typical Web documents, additional requests on a persistent connection normally do not arrive until after the response to the first request is delivered to the client. For this reason, the front-end has to base its choice of a back-end node to handle the connection on knowledge of only the first request.

With respect to (2), our extended LARD policy adds two additional considerations when choosing a node to handle a request arriving on an already handed off persistent connection. First, as long as the utilization on the connection handling node's local disk is low, the content is read from that disk, avoiding the forwarding overhead. Second, in choosing a back-end to forward the request to, the policy only considers those nodes as candidates that currently cache the requested target.

Regarding (3), the extended LARD policy uses a simple heuristic to decide whether content should be cached on the connection handling node. When the disk utilization on the connection handling node is high, it is assumed that the node's main memory cache is already thrashing. Therefore, the requested content is not cached locally. If the disk utilization is low, then the requested content is added to the node's cache.

We now present the extended LARD policy. When the first request arrives on a persistent connection, the connection handling node is chosen using the basic LARD policy described in Section 4.1. For each subsequent request on the persistent connection:

- If the target is cached at the connection handling node or if the disk utilization on the connection handling node is low (less than 5 queued disk events), then the request is assigned to the same.
- Else, the three cost metrics presented in Section 4.1 are computed over the connection handling node and any other back-end nodes that have the target cached. The request is then assigned to the node that yields the minimum aggregate cost.

For the purpose of computing the LARD cost metrics, a single load unit is assigned to the connec-

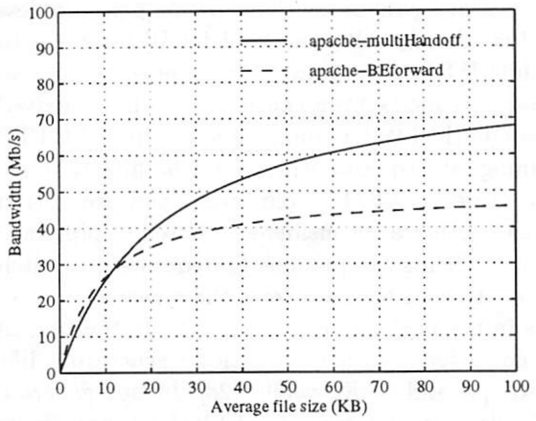


Figure 5: Apache

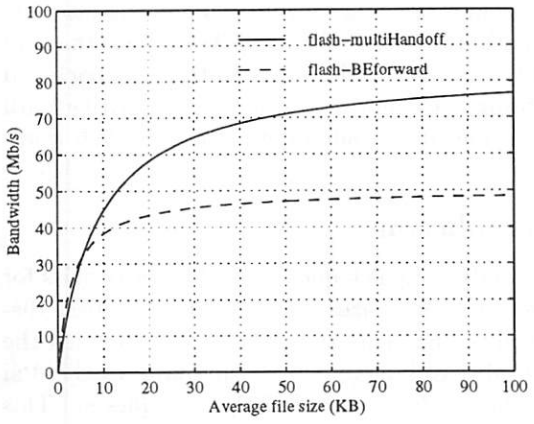


Figure 6: Flash

tion handling node for each active connection that it handles. When the back-end forwarding mechanism is used to fetch documents from other nodes, every such node is additionally assigned a load of 1/N units—where N is the number of outstanding requests in a batch of pipelined HTTP/1.1 requests—for the duration of the request handling of all N requests.

Ideally, the front-end should assign a load of 1 to a remote node during the service time of a request. However, the front-end cannot determine when exactly a HTTP/1.1 request is being served; it can, however, estimate the service time for a batch of N pipelined HTTP/1.1 requests. Therefore, it assigns a load of 1/N to each remote node for the entire batch service time.

The front-end estimates N as the number of requests in the last batch of closely spaced requests that arrived on the connection and it estimates the batch service time as the time it takes until the next batch arrives or the connection goes idle[3]. That is, the front-end assumes that all previous requests have finished once a new batch of requests arrives on the same connection.

As in LARD, mappings between targets and back-end nodes are updated each time a target is fetched from a back-end node. It is to be noted that the extended LARD policy is equivalent to LARD for HTTP/1.0 requests.

5 Performance Analysis of Distribution Mechanisms

This section presents a simple analysis of the fundamental performance tradeoff in the use of the multiple handoff mechanism versus the back-end forwarding mechanism for request distribution in the presence of persistent connections.

When compared to the multiple handoff mechanism, the back-end forwarding mechanism trades off a per-byte response forwarding cost for a per-request handoff overhead. This would suggest that back-end request forwarding might be most appropriate for requests that result in small amounts of response data, while the multiple handoff approach should win in case of large responses, assuming that all other factors that affect performance are equal.

Figures 5 and 6 show the results of a simple analysis that confirms and quantifies this intuition. The analysis predicts the server bandwidth, as a function of average response size, that can be obtained from a cluster with four nodes, using either the multiple handoff or the back-end forwarding mechanism. The analysis is based on the values for handoff overhead, per-request overhead, and per-byte forwarding overhead reported above for the Apache and Flash Web servers, respectively.

To expose the full impact of the mechanisms, pessimal assumptions are made with respect to the request distribution policy. It is assumed that all requests after the first one arriving on a persistent connection have to be served by a back-end node other than the connection handling node. Since most practical policies can do better than this, the results indicate an upper bound on the impact of the choice of the request distribution mechanism on the actual cluster performance.

The results confirm that for small response sizes, the back-end forwarding mechanism yields higher performance, while the multiple handoff mechanism is superior for large responses. The crossover point depends on the relative cost of handoff versus data forwarding, and lies at 12 KB for Apache and 6 KB for Flash. These results are nearly independent of

[3] An idle connection can be detected at the front-end by the absence of ACKs from the client.

the average number of requests received on a persistent connection. Since the average response size in today's HTTP/1.0 Web traffic is less than 13 KB [19, 3], these results indicate that the back-end forwarding mechanism is indeed competitive with the TCP multiple handoff mechanism on Web workloads.

6 Simulation

To study various request distribution policies for a range of cluster sizes using different request distribution mechanisms and policies, we extended the configurable Web server cluster simulator used in Pai et al. [23] to deal with HTTP/1.1 requests. This section gives an overview of the simulator. A more detailed description of the simulator can be found in Pai et al. [23].

The costs for the basic request processing steps used in our simulations were derived by performing measurements on a 300 MHz Pentium II machine running FreeBSD 2.2.6 and either the widely used Apache 1.3.3 Web server, or an aggressively optimized research Web server called Flash [24, 25]. Connection establishment and teardown costs are set at 278/129 μs of CPU time each, per-request overheads at 527/159 μs, and transmit processing incurs 24/24 μs per 512 bytes to simulate Apache/Flash, respectively.

Using these numbers, an 8 KByte document can be served from the main memory cache at a rate of approximately 682/1248 requests/sec with Apache/Flash, respectively, using HTTP/1.0 connections. The rate is higher for HTTP/1.1 connections and depends upon the average number of requests per connection. The back-end machines used in our prototype implementation have a main memory size of 128 MB. However, the main memory is shared between the OS kernel, server applications and file cache. To account for this, we set the back-end cache size in our simulations to 85 MB.

The simulator does not model TCP behavior for the data transmission. For example, the data transmission is assumed to be continuous rather than limited by the TCP slow-start [29]. This does not affect the throughput results as networks are assumed to be infinitely fast and thus throughput is limited only by the disk and CPU overheads.

The workload used by the simulator was derived from logs of actual Web servers. The logs contain the name and the size of requested targets as well as the client host and the timestamp of the access. Unfortunately, most Web servers do not record whether two requests arrived on the same connection. To construct a simulator working with HTTP/1.1 requests, we used the following heuristic. Any set of requests sent by the same client with a period of less than 15s (the default time used by Web servers to close idle HTTP/1.1 connections) between any two successive requests were considered to have arrived on a single HTTP/1.1 connection. To model HTTP pipelining, all requests other than the first that are in the same HTTP/1.1 connection and are within 5s of each other are considered a batch of pipelined requests. Clients can pipeline all requests in a batch but have to wait for data from the server before requests in the next batch can be sent. To the best of our knowledge, synthetic workload generators like SURGE [4] and SPECweb96 [28] do not generate workloads representative of HTTP/1.1 connections.

The workload was generated by combining logs from multiple departmental Web servers at Rice University. This trace spans a two-month period. The same logs were used for generating the workload used in Pai et al. [23]. The data set for our trace consists of 31,000 targets covering 1.015 GB of space. Our results show that this trace needs 526/619/745 MB of memory to cover 97/98/99% of all requests, respectively.

The simulator calculates overall throughput, cache hit rate, average CPU and disk idle times at the back-end nodes, and other statistics. Throughput is the number of requests in the trace that were served per second by the entire cluster, calculated as the number of requests in the trace divided by the simulated time it took to finish serving all the requests in the trace. The request arrival rate was matched to the aggregate throughput of the server.

6.1 Simulation Results

In this section we present simulation results comparing the following mechanisms/policy combinations.

1. TCP single handoff with LARD on HTTP/1.0 workload [simple-LARD]
2. TCP single handoff with LARD on HTTP/1.1 workload [simple-LARD-PHTTP]
3. TCP multiple handoff with extended LARD on HTTP/1.1 workload [multiHandoff-extLARD-PHTTP]
4. Back-end forwarding with extended LARD on HTTP/1.1 workload [BEforward-extLARD-PHTTP]
5. Ideal handoff with extended LARD on HTTP/1.1 workload [zeroCost-extLARD-PHTTP]

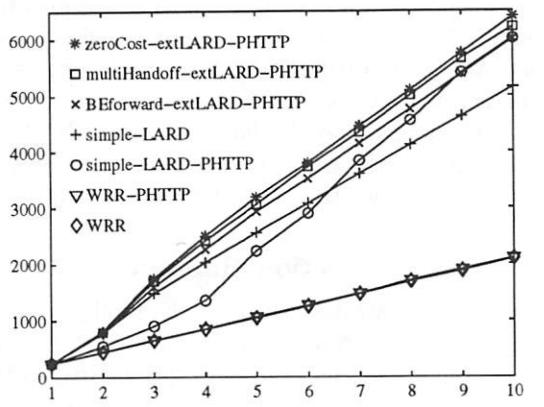


Figure 7: Apache Throughput

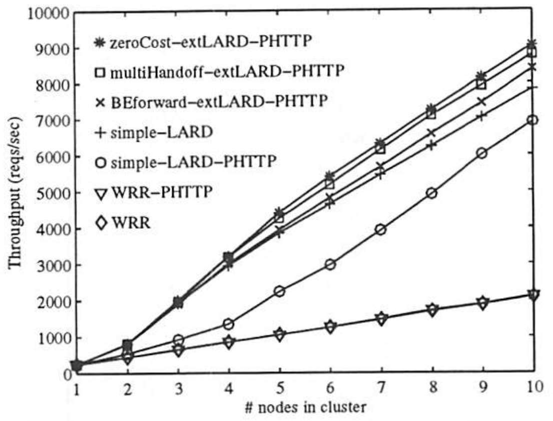


Figure 8: Flash Throughput

Most of these mechanisms have already been described in Section 3. The "ideal handoff" is an idealized mechanism that incurs no overhead for reassigning a persistent connection to another back-end node. It is useful as a benchmark, as performance results with this mechanism provide a ceiling for results that can be obtained with any practical request distribution mechanism.

Figures 7 and 8 show the throughput results with the Apache and Flash Web servers, respectively, running on the back-end nodes. For comparison, results for the widely used Weighted Round-Robin (WRR) policy are also included, on HTTP/1.0 and HTTP/1.1 workloads.

When driving simple LARD with a HTTP/1.1 workload (simple-LARD-PHTTP), results show that the throughput suffers considerably (up to 39% with Apache and up to 54% with Flash), particularly at small to medium cluster sizes. The loss of locality more than offsets the reduced server overhead of persistent connections.

The key result, however, is that the extended LARD policy both with the multiple handoff mechanism and the back-end forwarding mechanism (multiHandoff-extLARD-PHTTP and BEforward-extLARD-PHTTP) are within 8% of the ideal mechanism and afford throughput gains of up to 20% when compared to simple-LARD. Moreover, the throughput achieved with each mechanism is within 6%, confirming that both mechanisms are competitive on today's Web workloads.

The performance of LARD with HTTP/1.1 (simple-LARD-PHTTP) catches up with that of the extended LARD schemes for larger clusters. The reason is as follows. With a sufficient number of back-end nodes, the aggregate cache size of the cluster becomes much larger than the working set, allowing each back-end to cache not only the targets assigned to it by the LARD policy, but also additional targets requested in HTTP/1.1 connections. Eventually, enough targets are cached in each back-end node to yield high cache hit rates not only for the first request in a HTTP/1.1 connection, but also for subsequent requests. As a result, the performance approaches (but cannot exceed) that of the extended LARD strategies for large cluster sizes.

WRR cannot obtain throughput advantages from the use of persistent connections on our workload, as it remains disk bound for all cluster sizes and is therefore unable to capitalize on the reduced CPU overhead of persistent connections. As previously reported [23], simple-LARD outperforms WRR by a large margin as the cluster size increases, because it can aggregate the node caches. With one server node, the performance with HTTP/1.1 is identical to HTTP/1.0, because the back-end servers are disk bound with all policies.

The results obtained with the Flash Web server, which are likely to predict future trends in Web server software performance, differ mainly in that the performance loss of simple-LARD-PHTTP is more significant than with Apache. This underscores the importance of an efficient mechanism for handling persistent connections in cluster servers with content-based request distribution.

The throughput gains afforded by the hypothetical ideal handoff mechanism might also be achievable by a powerful relaying front-end (see Section 3.1) *as long as* it is not a bottleneck. However, as shown in Figures 7 and 8, such a front-end achieves only 8% better throughput than the back-end forwarding mechanism used with the extended LARD policy.

7 Prototype Cluster Design

This section describes the design of our prototype cluster. Given the complexity of the TCP mul-

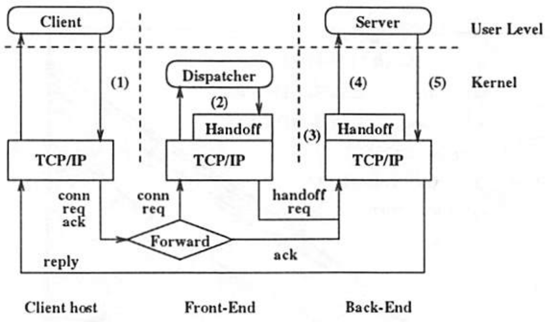


Figure 9: TCP connection handoff

tiple handoff mechanism, and the fact that simulation results indicate no substantial performance advantages of multiple handoff over back-end request forwarding, we decided to implement the back-end forwarding mechanism in the prototype.

Section 7.1 gives an overview of the various components of the cluster. Section 7.2 describes the TCP single handoff protocol. Section 7.3 describes *tagging*, a technique by which the front-end instructs the connection handling node to forward a given request to another back-end node. In Section 7.4, we describe how the back-end nodes fetch requests remotely from other nodes in a manner that keeps the server applications unchanged.

7.1 Overview

The cluster consists of a front-end node connected to the back-end nodes with a high-speed LAN. HTTP clients are not aware of the existence of the back-end nodes, and the cluster effectively provides the illusion of a single Web server machine to the clients.

Figure 9 shows the user-level processes and protocol stacks at the client, the front-end and the back-ends. The client application (e.g., Web browser) is unchanged and runs on an unmodified standard operating system. The server process at the back-end machines is also unchanged, and can be any off-the-shelf Web server application (e.g., Apache [2], Zeus [31]). The front-end and back-end protocol stacks, however, employ some additional components, which are added via a loadable kernel module.

The front-end and back-end nodes use the TCP single handoff protocol, which runs over the standard TCP/IP to provide a control session between the front-end and the back-end machine. The LARD and extended LARD policies are implemented in a *dispatcher* module at the front-end. In addition, the front-end also contains a *forwarding module*, which will be described in Section 7.2. The front-end and back-end nodes also have a user-level startup process (not shown in Figure 9) that is used to initialize the dispatcher and setup the control sessions between the front-end and the back-end handoff protocols. After initializing the cluster, these processes remain kernel resident and provide a process context for the dispatcher and the handoff protocols. Disk queue lengths at the back-end nodes are conveyed to the front-end using the control sessions mentioned above.

7.2 TCP Connection Handoff

Figure 9 illustrates a typical handoff: (1) the client process (e.g., Netscape) uses the TCP/IP protocol to connect to the front-end, (2) the *dispatcher* module at the front-end accepts the connection, and hands it off to a back-end using the TCP handoff protocol, (3) the back-end takes over the connection using its handoff protocol, (4) the server at the back-end accepts the created connection, and (5) the server at the back-end sends replies directly to the client.

The handoff remains transparent to the client in that all packets from the connection handling node appear to be coming from the front-end. All TCP packets from the client are forwarded by the front-end's forwarding module to the connection handling back-end. A copy of any packets containing requests from the client is sent up to the dispatcher to enable it to assign the requests to back-end nodes. HTTP/1.1 request pipelining [19, 21] is fully supported by the handoff protocol, and allows the clients to send multiple requests without waiting for responses from previous requests.

The TCP multiple handoff mechanism discussed in Section 3.2 can be implemented by extending the above design in the following manner. As soon as the back-end server at the connection-handling node indicates that it has sent all requisite data to the client, the handoff protocol at the back-end can *hand-back* the connection to the front-end that can further hand it to another back-end. Alternatively, the connection can be handed directly to another back-end after informing the front-end to forward future packets from the client appropriately. One of the main challenges in this design is to prevent the TCP pipeline from draining during the process of a handoff.

7.3 Tagging requests

As mentioned in the previous subsection, the forwarding module sends a copy of all request packets to the dispatcher once the connection has been handed off. Assignment of subsequent requests on the connection to back-end nodes other than the connection handling node is accomplished by *tagging*

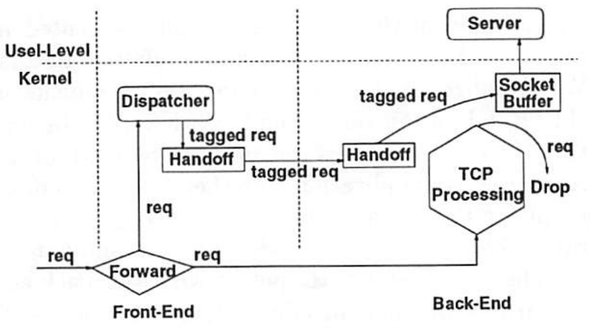


Figure 10: Tagging P-HTTP requests

the request content. The dispatcher sends these requests reliably to the connection handling back-end using the control session between the handoff protocol modules. The handoff protocol at the back-end receives the requests, and places them directly into the Web server's socket buffer. The tags enable the Web server to fetch the target using back-end forwarding (see Section 7.4). It remains, however, unaware of the presence of the handoff protocol.

After the handoff, all packets from the client are sent by the forwarding module to the connection handling node where they undergo TCP processing. Thus, after the handoff, data packets from the client are acknowledged by the connection handling node. The contents of these request packets, once received, are however discarded by the connection handling node (see Figure 10). Instead, the tagged requests received from the front-end via the control connection are delivered to the server process.

7.4 Fetching remote requests

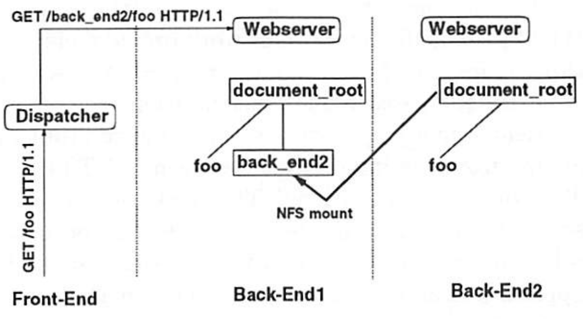


Figure 11: Transparent remote request fetching

Web server applications typically serve documents from a user-configurable directory which we will refer to as *document_root*. To implement remote fetching transparently to the Web server application, each back-end node NFS mounts the document_root from other back-end nodes on a subdirectory in its own document_root directory. Tagging is accomplished by the front-end dispatcher changing the URL in the client requests by prepending the name of the directory corresponding to the remote back-end node. Figure 11 depicts the situation where the dispatcher tags an HTTP GET request by prepending *back-end2* to the URL in order to make *back-end1* fetch file *foo* using NFS.

An issue concerning the fetching of remote files is NFS client caching, which would result in caching of targets at multiple back-end nodes and interfere with LARD's ability to control cache replication. To avoid this problem, we made a small modification in FreeBSD to disable client side caching of NFS files.

8 Prototype Cluster Performance

In this section, we present performance results obtained with a prototype cluster.

8.1 Experimental Environment

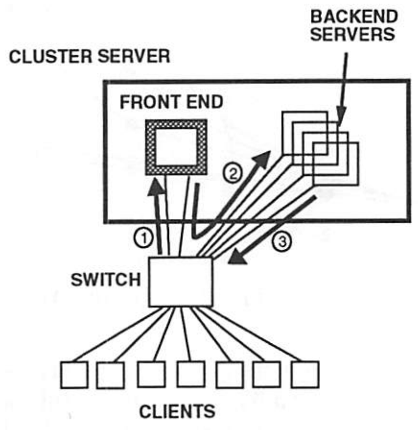


Figure 12: Experimental Testbed

Our testbed consists of a number of client machines connected to a cluster server. The configuration is shown in Figure 12. Traffic from the clients flows to the front-end (1) and is forwarded to the back-ends (2). Data packets transmitted from the back-ends to the clients bypass the front-end (3).

The front-end of the server cluster is a 300MHz Intel Pentium II based PC with 128MB of memory. The cluster back-end consists of six PCs of the same type and configuration as the front-end. All machines run FreeBSD 2.2.6. A loadable kernel module was added to the OS of the front-end and back-end nodes that implements the TCP single handoff protocol, and, in the case of the front-end, the forwarding module. The clients are seven 166MHz Intel Pentium Pro PCs, each with 64MB of memory.

The clients and back-end nodes in the cluster are connected using switched Fast Ethernet (100Mbps). The front-end and the back-end nodes are equipped with two network interfaces, one for communication with the clients, one for internal communication. Clients, front-end, and back-ends are connected through a single 24-port switch. All network interfaces are Intel EtherExpress Pro/100B running in full-duplex mode.

The Apache-1.3.3 [2] server was used on the back-end nodes. Our client software is an event-driven program that simulates multiple HTTP clients. Each simulated HTTP client makes HTTP requests as fast as the server cluster can handle them.

8.2 Cluster Performance Results

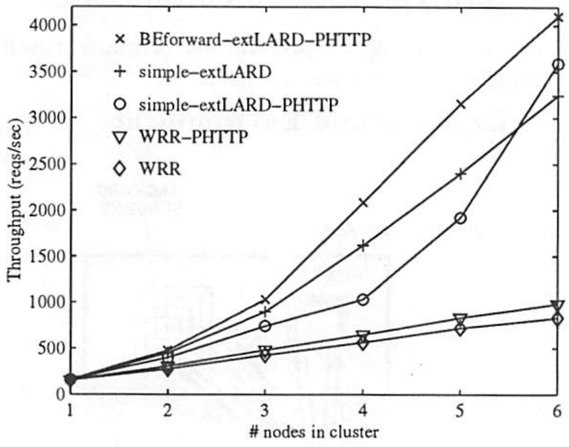


Figure 13: HTTP Throughput (Apache)

We used a segment of the Rice University trace alluded to in Section 6 to drive our prototype cluster. A single back-end node running Apache 1.3.3 can deliver about 151 req/s on this trace.

The Apache Web server relies on the file caching services of the underlying operating system. FreeBSD uses a unified buffer cache, where cached files are competing with user processes for physical memory pages. All page replacement is controlled by FreeBSD's pageout daemon, which implements a variant of the clock algorithm [18]. The cache size is variable, and depends on main memory pressure from user applications. In our 128MB back-ends, memory demands from kernel and Apache server processes leave about 100MB of free memory. In practice, we observed file cache sizes between 70 and 95 MB.

The mechanism used for the WRR policy is similar to the simple TCP handoff in that the data from the back-end servers is sent directly to the clients. However, the assignment of connections to back-end nodes is purely load-based.

Several observations can be made from the results presented in Figure 13. The measurements largely confirm the simulation results presented in Section 6.1. Contrary to the simulation results, WRR realizes modest performance improvements on HTTP/1.1 on this disk-bound workload. We believe that HTTP/1.1 reduces the memory demands of the Apache server application, and therefore leaves more room for the file system cache, causing better hit rates. This effect is not modeled by our simulator.

The extended LARD policy with the back-end forwarding mechanism affords four times as much throughput as WRR both with or without persistent connections and up to 26% better throughput with persistent connections than without. Without a mechanism for distributing HTTP/1.1 requests among back-end nodes, the LARD policies perform up to 35% worse in the presence of persistent connections.

Running extended LARD with the back-end forwarding mechanism and with six back-end nodes results in a CPU utilization of about 60% at the front-end. This indicates that the front-end can support 10 back-ends of equal CPU speed. Scalability to larger cluster sizes can be achieved by employing an SMP based front-end machine.

9 Related Work

Padmanabhan and Mogul [22] have shown that HTTP/1.0 connections can increase server resource requirements, the number of network packets per request, and effective latency perceived by the client. They proposed persistent connections and pipelining of HTTP requests, which have been adopted by the HTTP/1.1 standard [11]. The work in [19, 21] shows that these techniques dramatically improve HTTP/1.0 inefficiencies. Our work provides efficient support for HTTP/1.1 on cluster based Web servers with content-based request distribution.

Heidemann [13] describes performance problems arising from the interactions between P-HTTP and TCP in certain situations. The work also proposes some fixes that improve performance. The proposed solutions are complimentary to our work and can be applied in our cluster environment. In fact, most of the proposed fixes are already incorporated in Apache 1.3.3 [2].

Much current research addresses the scalability problems posed by the Web. The work includes co-operative caching proxies inside the network, push-based document distribution, and other innovative techniques [20, 7, 10, 16, 17, 27]. Our proposal addresses the complementary issue of providing support for HTTP/1.1 in cost-effective, scalable network servers.

Network servers based on clusters of workstations are starting to be widely used [12]. Several products are available or have been announced for use as front-end nodes in such cluster servers [8, 15]. To the best of our knowledge, the request distribution strategies used in the cluster front-ends are all variations of weighted round-robin, and do not take into account a request's target content. An exception is the Dispatch product by Resonate, Inc., which supports content-based request distribution [26]. The product does not appear to use any dynamic distribution policies based on content and no attempt is made to achieve cache aggregation via content-based request distribution.

Hunt et al. proposed a TCP option designed to enable content-based load distribution in a cluster server [14]. The design is roughly comparable in functionality to our TCP single handoff protocol, but has not been implemented.

Fox et al. [12] report on the cluster server technology used in the Inktomi search engine. The work focuses on the reliability and scalability aspects of the system and is complementary to our work. The request distribution policy used in their systems is based on weighted round-robin.

Loosely-coupled distributed servers are widely deployed on the Internet. Such servers use various techniques for load balancing including DNS round-robin [6], HTTP client re-direction [1], Smart clients [30], source-based forwarding [9] and hardware translation of network addresses [8]. Some of these schemes have problems related to the quality of the load balance achieved and the increased request latency. A detailed discussion of these issues is made in the work by Goldszmidt and Hunt [15] and Damani et al. [9]. None of these schemes support content-based request distribution.

10 Conclusions

Persistent connections pose problems for cluster based Web servers that use content-based request distribution, because requests that appear in a single connection may have to be served by different back-end nodes. We describe two efficient mechanisms for distributing requests arriving on persistent connections, TCP multiple handoff and back-end request forwarding.

A simulation study shows that both mechanisms can efficiently handle Web workloads on persistent connections. Moreover, we extend the *locality aware request distribution* (LARD) strategy to work with back-end request forwarding and show that it yields performance that is within 8% of results obtained with a simulated idealized mechanism. The proposed policies and mechanisms are fully transparent to the HTTP clients.

Finally, we have implemented the extended LARD policy and the back-end request forwarding mechanism in a prototype cluster. Performance results indicate that the extended LARD strategy affords up to 26% improvement in throughput with persistent connections over HTTP/1.0. Our results also indicate that a single front-end CPU can support up to 10 back-end nodes of equal speed.

In this paper, we have focused on studying HTTP servers that serve static content. Further research is needed for supporting request distribution mechanisms and policies for dynamic content.

11 Acknowledgments

We would like to thank Erich Nahum and the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by NSF Grants CCR-9803673, CCR-9503098, MIP-9521386, by Texas TATP Grant 003604, and by an IBM Partnership Award.

References

[1] D. Andresen et al. SWEB: Towards a Scalable WWW Server on MultiComputers. In *Procceedings of the 10th International Parallel Processing Symposium*, Apr. 1996.

[2] Apache. http://www.apache.org/.

[3] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, Apr. 1996.

[4] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the ACM SIGMETRICS Conference*, Madison, WI, July 1998.

[5] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext transfer protocol – HTTP/1.0, May 1996.

[6] T. Brisco. DNS Support for Load Balancing. RFC 1794, Apr. 1995.

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 USENIX Technical Conference*, Jan. 1996.

[8] Cisco Systems Inc. LocalDirector. http://www.cisco.com.

[9] O. P. Damani, P.-Y. E. Chung, Y. Huang, C. Kintala, and Y.-M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems*, 29:1019–1027, 1997.

[10] P. Danzig, R. Hall, and M. Schwartz. A case for caching file objects inside internetworks. In *Proceedings of the ACM SIGCOMM '93 Conference*, Sept. 1993.

[11] R. Fielding, J. Gettys, J. Mogul, H. Nielsen, and T. Berners-Lee. RFC 2068: Hypertext transfer protocol – HTTP/1.1, Jan. 1997.

[12] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[13] J. Heidemann. Performance interactions between P-HTTP and TCP implementations. *ACM Computer Communication Review*, 27(2):65–73, April 1997.

[14] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, May 1997.

[15] IBM Corporation. IBM interactive network dispatcher. http://www.ics.raleigh.ibm.com/ics/isslearn.htm.

[16] T. M. Kroeger, D. D. Long, and J. C. Mogul. Exploring the bounds of Web latency reduction from caching and prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[17] G. R. Malan, F. Jahanian, and S. Subramanian. Salamander: A push-based distribution substrate for Internet applications. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[18] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, 1996.

[19] J. C. Mogul. The Case for Persistent-Connection HTTP. In *Proceedings of the ACM SIGCOMM '95 Symposium*, 1995.

[20] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM '97 Symposium*, Cannes, France, Sept. 1997.

[21] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of the ACM SIGCOMM '97 Symposium*, Cannes, France, Sept. 1997.

[22] V. N. Padmanabhan and J. C. Mogul. Improving HTTP Latency. In *Proceedings of the Second International WWW Conference*, Chicago, IL, Oct. 1994.

[23] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[24] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX Technical Conference*, Monterey, CA, June 1999.

[25] V. S. Pai, P. Druschel, and W. Zwaenepoel. I/O-Lite: A unified I/O buffering and caching system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[26] Resonate Inc. Resonate dispatch. http://www.resonateinc.com.

[27] M. Seltzer and J. Gwertzman. The Case for Geographical Pushcaching. In *Proceedings of the 1995 Workshop on Hot Operating Systems*, 1995.

[28] SPECWeb96. http://www.specbench.org/osg/web96/.

[29] W. Stevens. *TCP/IP Illustrated Volume 1 : The Protocols*. Addison-Wesley, Reading, MA, 1994.

[30] B. Yoshikawa et al. Using Smart Clients to Build Scalable Services. In *Proceedings of the 1997 USENIX Technical Conference*, Jan. 1997.

[31] Zeus. http://www.zeus.co.uk/.

Flash: An efficient and portable Web server

Vivek S. Pai[‡] Peter Druschel[†] Willy Zwaenepoel[†]
[‡] *Department of Electrical and Computer Engineering*
[†] *Department of Computer Science*
Rice University

Abstract

This paper presents the design of a new Web server architecture called the asymmetric multi-process event-driven (AMPED) architecture, and evaluates the performance of an implementation of this architecture, the Flash Web server. The Flash Web server combines the high performance of single-process event-driven servers on cached workloads with the performance of multi-process and multi-threaded servers on disk-bound workloads. Furthermore, the Flash Web server is easily portable since it achieves these results using facilities available in all modern operating systems.

The performance of different Web server architectures is evaluated in the context of a single implementation in order to quantify the impact of a server's concurrency architecture on its performance. Furthermore, the performance of Flash is compared with two widely-used Web servers, Apache and Zeus. Results indicate that Flash can match or exceed the performance of existing Web servers by up to 50% across a wide range of real workloads. We also present results that show the contribution of various optimizations embedded in Flash.

1 Introduction

The performance of Web servers plays a key role in satisfying the needs of a large and growing community of Web users. Portable high-performance Web servers reduce the hardware cost of meeting a given service demand and provide the flexibility to change hardware platforms and operating systems based on cost, availability, or performance considerations.

Web servers rely on caching of frequently-requested Web content in main memory to achieve throughput rates of thousands of requests per second, despite the long latency of disk operations. Since the data set size of Web workloads typically exceed the capacity of a server's main memory, a high-performance Web server must be structured such that it can overlap the serving of requests for cached content with concurrent disk operations that fetch requested content not currently cached in main memory.

Web servers take different approaches to achieving this concurrency. Servers using a *single-process event-driven (SPED)* architecture can provide excellent performance for cached workloads, where most requested content can be kept in main memory. The Zeus server [32] and the original Harvest/Squid proxy caches employ the SPED architecture[1].

On workloads that exceed that capacity of the server cache, servers with *multi-process (MP)* or *multi-threaded (MT)* architectures usually perform best. Apache, a widely-used Web server, uses the MP architecture on UNIX operating systems and the MT architecture on the Microsoft Windows NT operating system.

This paper presents a new portable Web server architecture, called asymmetric multi-process event-driven (AMPED), and describes an implementation of this architecture, the Flash Web server. Flash nearly matches the performance of SPED servers on cached workloads while simultaneously matching or exceeding the performance of MP and MT servers on disk-intensive workloads. Moreover, Flash uses only standard APIs and is therefore easily portable.

Flash's AMPED architecture behaves like a single-process event-driven architecture when requested documents are cached and behaves similar to a multi-process or multi-threaded architecture when requests must be satisfied from disk. We qualitatively and quantitatively compare the AMPED architecture to the SPED, MP, and MT approaches in the context of a single server implementation. Finally, we experimentally compare the performance of Flash to that of Apache and Zeus on real workloads obtained from server logs, and on two operating systems.

The rest of this paper is structured as follows: Section 2 explains the basic processing steps required of all Web servers and provides the background for

[1]Zeus can be configured to use multiple SPED processes, particularly when running on multiprocessor systems

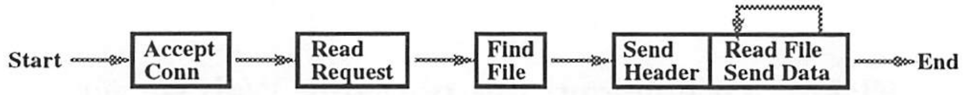


Figure 1: Simplified Request Processing Steps

the following discussion. In Section 3, we discuss the asynchronous multi-process event-driven (AMPED), the single-process event-driven (SPED), the multi-process (MP), and the multi-threaded (MT) architectures. We then discuss the expected architecture-based performance characteristics in Section 4 before discussing the implementation of the Flash Web server in Section 5. Using real and synthetic workloads, we evaluate the performance of all four server architectures and the Apache and Zeus servers in Section 6.

2 Background

In this section, we briefly describe the basic processing steps performed by an HTTP (Web) server. HTTP clients use the TCP transport protocol to contact Web servers and request content. The client opens a TCP connection to the server, and transmits a HTTP request header that specifies the requested content.

Static content is stored on the server in the form of disk files. *Dynamic content* is generated upon request by auxiliary application programs running on the server. Once the server has obtained the requested content, it transmits a HTTP response header followed by the requested data, if applicable, on the client's TCP connection.

For clarity, the following discussion focuses on serving HTTP/1.0 requests for static content on a UNIX-like operating system. However, all of the Web server architectures discussed in this paper are fully capable of handling dynamically-generated content. Likewise, the basic steps described below are similar for HTTP/1.1 requests, and for other operating systems, like Windows NT.

The basic sequential steps for serving a request for static content are illustrated in Figure 1, and consist of the following:

Accept client connection - accept an incoming connection from a client by performing an `accept` operation on the server's `listen` socket. This creates a new socket associated with the client connection.

Read request - read the HTTP request header from the client connection's socket and parse the header for the requested URL and options.

Find file - check the server filesystem to see if the requested content file exists and the client has appropriate permissions. The file's size and last modification time are obtained for inclusion in the response header.

Send response header - transmit the HTTP response header on the client connection's socket.

Read file - read the file data (or part of it, for larger files) from the filesystem.

Send data - transmit the requested content (or part of it) on the client connection's socket. For larger files, the "Read file" and "Send data" steps are repeated until all of the requested content is transmitted.

All of these steps involve operations that can potentially block. Operations that read data or accept connections from a socket may block if the expected data has not yet arrived from the client. Operations that write to a socket may block if the TCP send buffers are full due to limited network capacity. Operations that test a file's validity (using `stat()`) or open the file (using `open()`) can block until any necessary disk accesses complete. Likewise, reading a file (using `read()`) or accessing data from a memory-mapped file region can block while data is read from disk.

Therefore, a high-performance Web server must interleave the sequential steps associated with the serving of multiple requests in order to overlap CPU processing with disk accesses and network communication. The server's *architecture* determines what strategy is used to achieve this interleaving. Different server architectures are described in Section 3.

In addition to its architecture, the performance of a Web server implementation is also influenced by various optimizations, such as caching. In Section 5, we discuss specific optimizations used in the Flash Web server.

3 Server Architectures

In this section, we describe our proposed asymmetric multi-process event-driven (AMPED) architecture, as well as the existing single-process event-driven (SPED), multi-process (MP), and multi-threaded (MT) architectures.

3.1 Multi-process

In the multi-process (MP) architecture, a process is assigned to execute the basic steps associated with

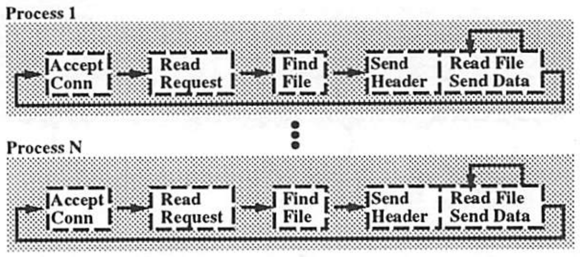


Figure 2: Multi-Process - In the MP model, each server process handles one request at a time. Processes execute the processing stages sequentially.

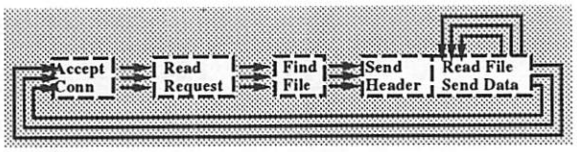


Figure 3: Multi-Threaded - The MT model uses a single address space with multiple concurrent threads of execution. Each thread handles a request.

serving a client request sequentially. The process performs all the steps related to one HTTP request before it accepts a new request. Since multiple processes are employed (typically 20-200), many HTTP requests can be served concurrently. Overlapping of disk activity, CPU processing and network connectivity occurs naturally, because the operating system switches to a runnable process whenever the currently active process blocks.

Since each process has its own private address space, no synchronization is necessary to handle the processing of different HTTP requests[2]. However, it may be more difficult to perform optimizations in this architecture that rely on global information, such as a shared cache of valid URLs. Figure 2 illustrates the MP architecture.

3.2 Multi-threaded

Multi-threaded (MT) servers, depicted in Figure 3, employ multiple independent threads of control operating within a single shared address space. Each thread performs all the steps associated with one HTTP request before accepting a new request, similar to the MP model's use of a process.

The primary difference between the MP and the MT architecture, however, is that all threads can share global variables. The use of a single shared address space lends itself easily to optimizations that rely on shared state. However, the threads must use some form of synchronization to control access to the shared data.

The MT model requires that the operating system provides support for kernel threads. That is, when one thread blocks on an I/O operation, other runnable threads within the same address space must remain eligible for execution. Some operating systems (e.g., FreeBSD 2.2.6) provide only user-level thread libraries without kernel support. Such systems cannot effectively support MT servers.

[2]Synchronization is necessary inside the OS to accept incoming connections, since the accept queue is shared

3.3 Single-process event-driven

The single-process event-driven (SPED) architecture uses a single event-driven server process to perform concurrent processing of multiple HTTP requests. The server uses non-blocking systems calls to perform asynchronous I/O operations. An operation like the BSD UNIX `select` or the System V `poll` is used to check for I/O operations that have completed. Figure 4 depicts the SPED architecture.

A SPED server can be thought of as a state machine that performs one basic step associated with the serving of an HTTP request at a time, thus interleaving the processing steps associated with many HTTP requests. In each iteration, the server performs a `select` to check for completed I/O events (new connection arrivals, completed file operations, client sockets that have received data or have space in their send buffers.) When an I/O event is ready, it completes the corresponding basic step and initiates the next step associated with the HTTP request, if appropriate.

In principle, a SPED server is able to overlap the CPU, disk and network operations associated with the serving of many HTTP requests, in the context of a single process and a single thread of control. As a result, the overheads of context switching and thread synchronization in the MP and MT architectures are avoided. However, a problem associated with SPED servers is that many current operating systems do not provide suitable support for asynchronous disk operations.

In these operating systems, non-blocking `read` and `write` operations work as expected on network sockets and pipes, but may actually block when used on disk files. As a result, supposedly non-blocking `read` operations on files may still block the caller while disk I/O is in progress. Both operating systems used in our experiments exhibit this behavior (FreeBSD 2.2.6 and Solaris 2.6). To the best of our knowledge, the same is true for most versions of UNIX.

Many UNIX systems provide alternate APIs that implement true asynchronous disk I/O, but these APIs are generally not integrated with the `select`

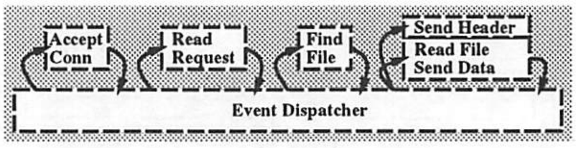


Figure 4: Single Process Event Driven - The SPED model uses a single process to perform all client processing and disk activity in an event-driven manner.

operation. This makes it difficult or impossible to simultaneously check for completion of network and disk I/O events in an efficient manner. Moreover, operations such as `open` and `stat` on file descriptors may still be blocking.

For these reasons, existing SPED servers do not use these special asynchronous disk interfaces. As a result, file `read` operations that do not hit in the file cache may cause the main server thread to block, causing some loss in concurrency and performance.

3.4 Asymmetric Multi-Process Event-Driven

The Asymmetric Multi-Process Event-Driven (AMPED) architecture, illustrated in Figure 5, combines the event-driven approach of the SPED architecture with multiple *helper* processes (or threads) that handle blocking disk I/O operations. By default, the main event-driven process handles all processing steps associated with HTTP requests. When a disk operation is necessary (e.g., because a file is requested that is not likely to be in the main memory file cache), the main server process instructs a *helper* via an inter-process communication (IPC) channel (e.g., a pipe) to perform the potentially blocking operation. Once the operation completes, the helper returns a notification via IPC; the main server process learns of this event like any other I/O completion event via `select`.

The AMPED architecture strives to preserve the efficiency of the SPED architecture on operations other than disk reads, but avoids the performance problems suffered by SPED due to inappropriate support for asynchronous disk reads in many operating systems. AMPED achieves this using only support that is widely available in modern operating systems.

In a UNIX system, AMPED uses the standard non-blocking `read`, `write`, and `accept` system calls on sockets and pipes, and the `select` system call to test for I/O completion. The `mmap` operation is used to access data from the filesystem and the `mincore` operation is used to check if a file is in main memory.

Note that the helpers can be implemented either as kernel threads within the main server process or

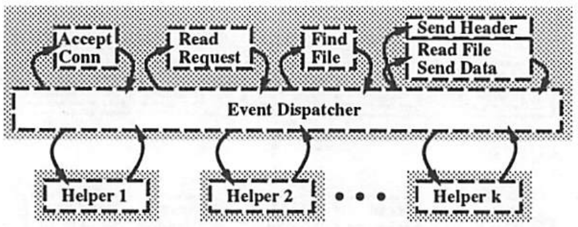


Figure 5: Asymmetric Multi-Process Event Driven - The AMPED model uses a single process for event-driven request processing, but has other helper processes to handle some disk operations.

as separate processes. Even when helpers are implemented as separate processes, the use of `mmap` allows the helpers to initiate the reading of a file from disk without introducing additional data copying. In this case, both the main server process and the helper `mmap` a requested file. The helper touches all the pages in its memory mapping. Once finished, it notifies the main server process that it is now safe to transmit the file without the risk of blocking.

4 Design comparison

In this section, we present a qualitative comparison of the performance characteristics and possible optimizations in the various Web server architectures presented in the previous section.

4.1 Performance characteristics

Disk operations - The cost of handling disk activity varies between the architectures based on what, if any, circumstances cause all request processing to stop while a disk operation is in progress. In the MP and MT models, only the process or thread that causes the disk activity is blocked. In AMPED, the helper processes are used to perform the blocking disk actions, so while they are blocked, the server process is still available to handle other requests. The extra cost in the AMPED model is due to the inter-process communication between the server and the helpers. In SPED, one process handles all client interaction as well as disk activity, so all user-level processing stops whenever any request requires disk activity.

Memory effects - The server's memory consumption affects the space available for the filesystem cache. The SPED architecture has small memory requirements, since it has only one process and one stack. When compared to SPED, the MT model incurs some additional memory consumption and kernel resources, proportional to the number of threads employed (i.e., the maximal number of concurrently served HTTP requests). AMPED's helper processes cause additional overhead, but the helpers have small application-level memory demands and a helper is needed only per concurrent disk operation, not for each concurrently served HTTP request. The MP model incurs the cost of a separate process per concurrently served HTTP request, which has substantial memory and kernel overheads.

Disk utilization - The number of concurrent disk requests that a server can generate affects whether it can benefit from multiple disks and disk head scheduling. The MP/MT models can cause one disk request per process/thread, while the AMPED model can generate one request per helper. In contrast, since all user-level processing stops in the SPED architecture whenever it accesses the disk, it can only generate one disk request at a time. As a result, it cannot benefit from multiple disks or disk head scheduling.

4.2 Cost/Benefits of optimizations & features

The server architecture also impacts the feasibility and profitability of certain types of Web server optimizations and features. We compare the tradeoffs necessary in the various architectures from a qualitative standpoint.

Information gathering - Web servers use information about recent requests for accounting purposes and to improve performance, but the cost of gathering this information across all connections varies in the different models. In the MP model, some form of interprocess communication must be used to consolidate data. The MT model either requires maintaining per-thread statistics and periodic consolidation or fine-grained synchronization on global variables. The SPED and AMPED architectures simplify information gathering since all requests are processed in a centralized fashion, eliminating the need for synchronization or interprocess communications when using shared state.

Application-level Caching - Web servers can employ application-level caching to reduce computation by using memory to store previous results, such as response headers and file mappings for frequently requested content. However, the cache memory competes with the filesystem cache for physical memory, so this technique must be applied carefully. In the MP model, each process may have its own cache in order to reduce interprocess communication and synchronization. The multiple caches increase the number of compulsory misses and they lead to less efficient use of memory. The MT model uses a single cache, but the data accesses/updates must be coordinated through synchronization mechanisms to avoid race conditions. Both AMPED and SPED can use a single cache without synchronization.

Long-lived connections - Long-lived connections occur in Web servers due to clients with slow links (such as modems), or through persistent connections in HTTP 1.1. In both cases, some server-side resources are committed for the duration of the connection. The cost of long-lived connections on the server depends on the resource being occupied. In AMPED and SPED, this cost is a file descriptor, application-level connection information, and some kernel state for the connection. The MT and MP models add the overhead of an extra thread or process, respectively, for each connection.

5 Flash implementation

The Flash Web server is a high-performance implementation of the AMPED architecture that uses aggressive caching and other techniques to maximize its performance. In this section, we describe the implementation of the Flash Web server and some of the optimization techniques used.

5.1 Overview

The Flash Web server implements the AMPED architecture described in Section 3. It uses a single non-blocking server process assisted by helper processes. The server process is responsible for all interaction with clients and CGI applications [26], as well as control of the helper processes. The helper processes are responsible for performing all of the actions that may result in synchronous disk activity. Separate processes were chosen instead of kernel threads to implement the helpers, in order to ensure portability of Flash to operating systems that do not (yet) support kernel threads, such as FreeBSD 2.2.6.

The server is divided into modules that perform the various request processing steps mentioned in Section 2 and modules that handle various caching functions. Three types of caches are maintained: filename translations, response headers, and file mappings. These caches and their function are explained below.

The helper processes are responsible for performing pathname translations and for bringing disk blocks into memory. These processes are dynamically spawned by the server process and are kept in reserve when not active. Each process operates synchronously, waiting on the server for new requests and handling only one request at a time. To minimize interprocess communication, helpers only return a completion notification to the server, rather than sending any file content they may have loaded from disk.

5.2 Pathname Translation Caching

The pathname translation cache maintains a list of mappings between requested filenames (e.g., "/~bob") and actual files on disk (e.g., /home/users/bob/public_html/index.html). This cache allows Flash to avoid using the pathname translation helpers for every incoming request. It reduces the processing needed for pathname translations, and it reduces the number of translation helpers needed by the server. As a result, the memory spent on the cache can be recovered by the reduction in memory used by helper processes.

5.3 Response Header Caching

HTTP servers prepend file data with a response header containing information about the file and the server, and this information can be cached and reused when the same files are repeatedly requested. Since the response header is tied to the underlying file, this cache does not need its own invalidation mechanism. Instead, when the mapping cache detects that a cached file has changed, the corresponding response header is regenerated.

5.4 Mapped Files

Flash retains a cache of memory-mapped files to reduce the number of map/unmap operations necessary for request processing. Memory-mapped files provide a convenient mechanism to avoid extra data copying and double-buffering, but they require extra system calls to create and remove the mappings. Mappings for frequently-requested files can be kept and reused, but unused mappings can increase kernel bookkeeping and degrade performance.

The mapping cache operates on "chunks" of files and lazily unmaps them when too much data has been mapped. Small files occupy one chunk each, while large files are split into multiple chunks. Inactive chunks are maintained in an LRU free list, and are unmapped when this list grows too large. We use LRU to approximate the "clock" page replacement algorithm used in many operating systems, with the goal of mapping only what is likely to be in memory. All mapped file pages are tested for memory residency via `mincore()` before use.

5.5 Byte Position Alignment

The `writev()` system call allows applications to send multiple discontiguous memory regions in one operation. High-performance Web servers use it to send response headers followed by file data. However, its use can cause misaligned data copying within the operating system, degrading performance. The extra cost for misaligned data is proportional to the amount of data being copied.

The problem arises when the OS networking code copies the various memory regions specified in a `writev` operation into a contiguous kernel buffer. If the size of the HTTP response header stored in the first region has a length that is not a multiple of the machine's word size, then the copying of all subsequent regions is misaligned.

Flash avoids this problem by aligning all response headers on 32-byte boundaries and padding their lengths to be a multiple of 32 bytes. It adds characters to variable length fields in the HTTP response header (e.g., the server name) to do the padding. The choice of 32 bytes rather than word-alignment is to target systems with 32-byte cache lines, as some systems may be optimized for copying on cache boundaries.

5.6 Dynamic Content Generation

The Flash Web server handles the serving of dynamic data using mechanisms similar to those used in other Web servers. When a request arrives for a dynamic document, the server forwards the request to the corresponding auxiliary (CGI-bin) application process that generates the content via a pipe. If a process does not currently exist, the server creates (e.g., forks) it.

The resulting data is transmitted by the server just like static content, except that the data is read from a descriptor associated with the CGI process' pipe, rather than a file. The server process allows the CGI application process to be persistent, amortizing the cost of creating the application over multiple requests. This is similar to the FastCGI [27] interface and it provides similar benefits. Since the CGI applications run in separate processes from the server, they can block for disk activity or other reasons and perform arbitrarily long computations without affecting the server.

5.7 Memory Residency Testing

Flash uses the `mincore()` system call, which is available in most modern UNIX systems, to determine if mapped file pages are memory resident. In operating systems that don't support this operation but provide the `mlock()` system call to lock memory pages (e.g., Compaq's Tru64 UNIX, formerly Digital Unix), Flash could use the latter to control its file cache management, eliminating the need for memory residency testing.

Should no suitable operations be available in a given operating system to control the file cache or test for memory residency, it may be possible to use a feedback-based heuristic to minimize blocking on disk I/O. Here, Flash could run the clock algorithm to predict which cached file pages are memory resident. The prediction can adapt to changes in the amount of memory available to the file cache by using continuous feedback from performance counters that keep track of page faults and/or associated disk accesses.

6 Performance Evaluation

In this section, we present experimental results that compare the performance of the different Web server architectures presented in Section 3 on real workloads. Furthermore, we present comparative performance results for Flash and two state-of-the-art Web servers, Apache [1] and Zeus [32], on synthetic and real workloads. Finally, we present results that quantify the performance impact of the various performance optimizations included in Flash.

To enable a meaningful comparison of different architectures by eliminating variations stemming from implementation differences, the same Flash code base is used to build four servers, based on the AMPED (Flash), MT (Flash-MT), MP (Flash-MP), and SPED (Flash-SPED) architectures. These four servers represent all the architectures discussed in this paper, and they were developed by replacing Flash's event/helper dispatch mechanism with the suitable counterparts in the other architectures. In all other respects, however, they are identical to the standard, AMPED-based version of Flash and use the same techniques and optimizations.

In addition, we compare these servers with two widely-used production Web servers, Zeus v1.30 (a high-performance server using the SPED architecture), and Apache v1.3.1 (based on the MP architecture), to provide points of reference.

In our tests, the Flash-MP and Apache servers use 32 server processes and Flash-MT uses 64 threads. Zeus was configured as a single process for the experiments using synthetic workloads, and in a two-process configuration advised by Zeus for the real workload tests. Since the SPED-based Zeus can block on disk I/O, using multiple server processes can yield some performance improvements even on a uniprocessor platform, since it allows the overlapping of computation and disk I/O.

Both Flash-MT and Flash use a memory-mapped file cache with a 128 MB limit and a pathname cache limit of 6000 entries. Each Flash-MP process has a mapped file cache limit of 4 MB and a pathname cache of 200 entries. Note that the caches in an MP server have to be configured smaller, since they are replicated in each process.

The experiments were performed with the servers running on two different operating systems, Solaris 2.6 and FreeBSD 2.2.6. All tests use the same server hardware, based on a 333 MHz Pentium II CPU with 128 MB of memory and multiple 100 Mbit/s Ethernet interfaces. A switched Fast Ethernet connects the server machine to the client machines that generate the workload. Our client software is an event-driven program that simulates multiple HTTP clients [3]. Each simulated HTTP client makes HTTP requests as fast as the server can handle them.

6.1 Synthetic Workload

In the first experiment, a set of clients repeatedly request the same file, where the file size is varied in each test. The simplicity of the workload in this test allows the servers to perform at their highest capacity, since the requested file is cached in the server's main memory. The results are shown in Figures 6 (Solaris) and 7 (FreeBSD). The left-hand side graphs plot the servers' total output bandwidth against the requested file size. The connection rate for small files is shown separately on the right.

Results indicate that the choice of architecture has little impact on a server's performance on a trivial, cached workload. In addition, the Flash variants compare favorably to Zeus, affirming the absolute performance of the Flash-based implementation. The Apache server achieves significantly lower performance on both operating systems and over the entire range of file sizes, most likely the result of the more aggressive optimizations employed in the Flash versions and presumably also in Zeus.

Flash-SPED slightly outperforms Flash because the AMPED model tests the memory-residency of files before sending them. Slight lags in the performance of Flash-MT and Flash-MP are likely due to the extra kernel overhead (context switching, etc.) in these architectures. Zeus' anomalous behavior on FreeBSD for file sizes between 10 and 100 KB appears to stem from the byte alignment problem mentioned in Section 5.5.

All servers enjoy substantially higher performance when run under FreeBSD as opposed to Solaris. The relative performance of the servers is not strongly affected by the operating system.

6.2 Trace-based experiments

While the single-file test can indicate a server's maximum performance on a cached workload, it gives little indication of its performance on real workloads. In the next experiment, the servers are subjected to a more realistic load. We generate a

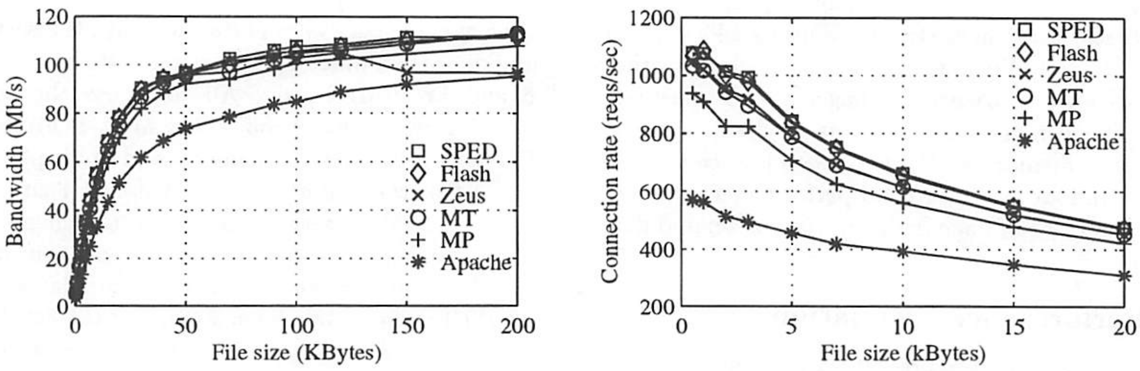


Figure 6: Solaris single file test — On this trivial test, server architecture seems to have little impact on performance. The aggressive optimizations in Flash and Zeus cause them to outperform Apache.

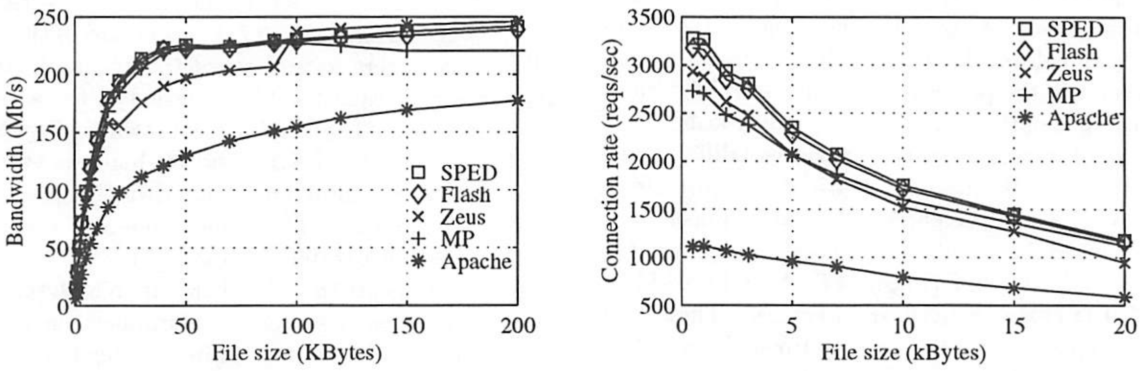


Figure 7: FreeBSD single file test — The higher network performance of FreeBSD magnifies the difference between Apache and the rest when compared to Solaris. The shape of the Zeus curve between 10 kBytes and 100 kBytes is likely due to the byte alignment problem mentioned in Section 5.5

client request stream by replaying access logs from existing Web servers.

Figure 8 shows the throughput in Mb/sec achieved with various Web servers on two different workloads. The "CS trace" was obtained from the logs of Rice University's Computer Science departmental Web server. The "Owlnet trace" reflects traces obtained from a Rice Web server that provides personal Web pages for approximately 4500 students and staff members. The results were obtained with the Web servers running on Solaris.

The results show that Flash with its AMPED architecture achieves the highest throughput on both workloads. Apache achieves the lowest performance. The comparison with Flash-MP shows that this is only in part the result of its MP architecture, and mostly due to its lack of aggressive optimizations like those used in Flash.

The Owlnet trace has a smaller dataset size than the CS trace, and it therefore achieves better cache locality in the server. As a result, Flash-SPED's relative performance is much better on this trace, while MP performs well on the more disk-intensive CS trace. Even though the Owlnet trace has high locality, its average transfer size is smaller than the CS trace, resulting in roughly comparable bandwidth numbers.

A second experiment evaluates server performance under realistic workloads with a range of dataset sizes (and therefore working set sizes). To generate an input stream with a given dataset size, we use the access logs from Rice's ECE departmental Web server and truncate them as appropriate to achieve a given dataset size. The clients then replay this truncated log as a loop to generate requests. In both experiments, two client machines with 32 clients each are used to generate the workload.

Figures 9 (BSD) and 10 (Solaris) shows the performance, measured as the total output bandwidth, of the various servers under real workload and various

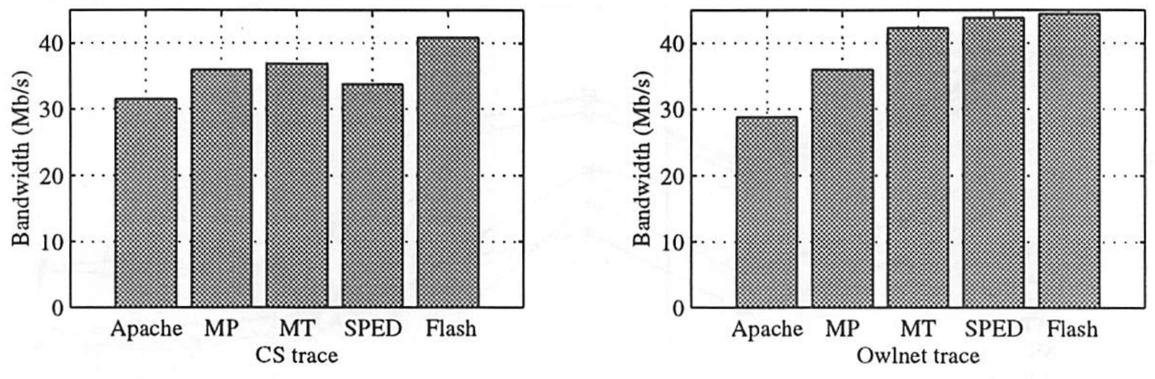


Figure 8: Performance on Rice Server Traces/Solaris

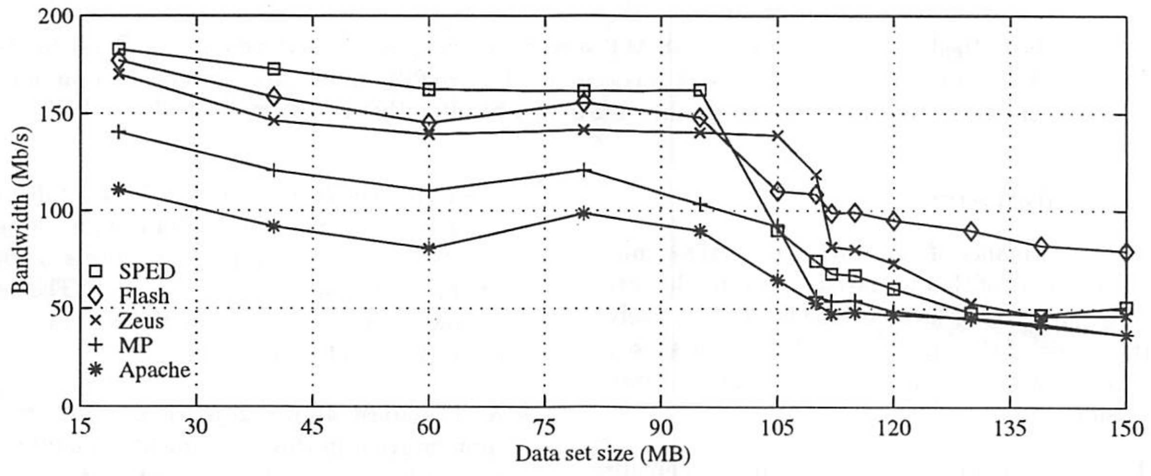


Figure 9: FreeBSD Real Workload - The SPED architecture is ideally suited for cached workloads, and when the working set fits in cache, Flash mimics Flash-SPED. However, Flash-SPED's performance drops drastically when operating on disk-bound workloads.

dataset sizes. We report output bandwidth instead of request/sec in this experiment, because truncating the logs at different points to vary the dataset size also changes the size distribution of requested content. This causes fluctuations in the throughput in requests/sec, but the output bandwidth is less sensitive to this effect.

The performance of all the servers declines as the dataset size increases, and there is a significant drop at the point when the working set size (which is related to the dataset size) exceeds the server's effective main memory cache size. Beyond this point, the servers are essentially disk bound. Several observation can be made based on these results:

- Flash is very competitive with Flash-SPED on cached workloads, and at the same time exceeds or meets the performance of the MP servers on disk-bound workloads. This confirms that Flash with its AMPED architecture is able to combine the best of other architectures across a wide range of workloads. This goal was central to the design of the AMPED architecture.

- The slight performance difference between Flash and Flash-SPED on the cached workloads reflects the overhead of checking for cache residency of requested content in Flash. Since the data is already in memory, this test causes unnecessary overhead on cached workloads.

- The SPED architecture performs well for cached workloads but its performance deteriorates quickly as disk activity increases. This confirms our earlier reasoning about the performance tradeoffs associated with this architecture. The same behavior can be seen in the SPED-based Zeus' performance, although its absolute performance falls short of the various

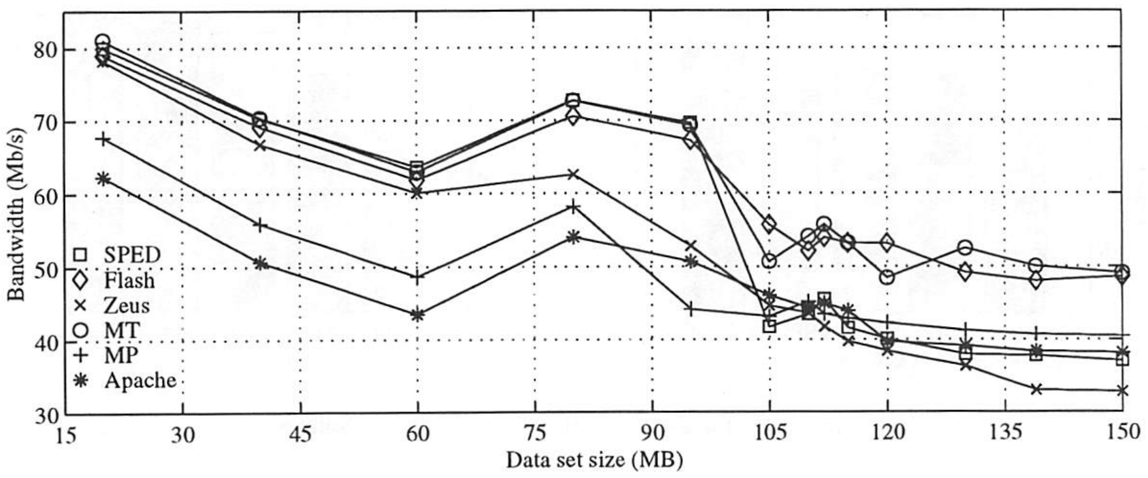


Figure 10: Solaris Real Workload - The Flash-MT server has comparable performance to Flash for both in-core and disk-bound workloads. This result was achieved by carefully minimizing lock contention, adding complexity to the code. Without this effort, the disk-bound results otherwise resembled Flash-SPED.

Flash-derived servers.

- The performance of Flash MP server falls significantly short of that achieved with the other architectures on cached workloads. This is likely the result of the smaller user-level caches used in Flash-MP as compared to the other Flash versions.

- The choice of an operating system has a significant impact on Web server performance. Performance results obtained on Solaris are up to 50% lower than those obtained on FreeBSD. The operating system also has some impact on the relative performance of the various Web servers and architectures, but the trends are less clear.

- Flash achieves higher throughput on disk-bound workloads because it can be more memory-efficient and causes less context switching than MP servers. Flash only needs enough helper processes to keep the disk busy, rather than needing a process per connection. Additionally, the helper processes require little application-level memory. The combination of fewer total processes and small helper processes reduces memory consumption, leaving extra memory for the filesystem cache.

- The performance of Zeus on FreeBSD appears to drop only after the data set exceeds 100 MB, while the other servers drop earlier. We believe this phenomenon is related to Zeus's request-handling, which appears to give priority to requests for small documents. Under full load, this tends to starve requests for large documents and thus causes the server to process a somewhat smaller effective working set. The overall lower performance under Solaris appears to mask this effect on that OS.

- As explained above, Zeus uses a two-process configuration in this experiment, as advised by the vendor. It should be noted that this gives Zeus a slight advantage over the single-process Flash-SPED, since one process can continue to serve requests while the other is blocked on disk I/O.

Results for the Flash-MT servers could not be provided for FreeBSD 2.2.6, because that system lacks support for kernel threads.

6.3 Flash Performance Breakdown

The next experiment focuses on the Flash server and measures the contribution of its various optimizations on the achieved throughput. The configuration is identical to the single file test on FreeBSD, where clients repeatedly request a cached document of a given size. Figure 11 shows the throughput obtained by various versions of Flash with all combinations of the three main optimizations (pathname translation caching, mapped file caching, and response header caching).

The results show that each of the optimizations has a significant impact on server throughput for cached content, with pathname translation caching providing the largest benefit. Since each of the op-

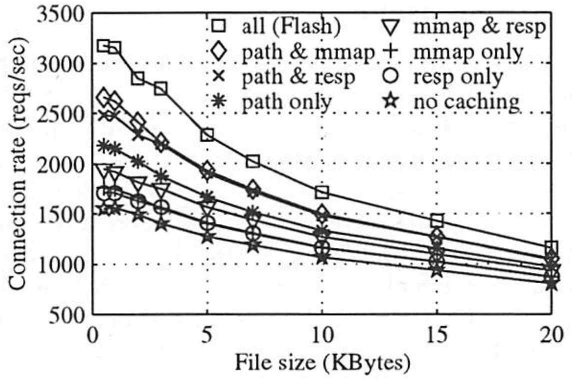


Figure 11: Flash Performance Breakdown - Without optimizations, Flash's small-file performance would drop in half. The eight lines show the effect of various combinations of the caching optimizations.

timization avoids a per-request cost, the impact is strongest on requests for small documents.

6.4 Performance under WAN conditions

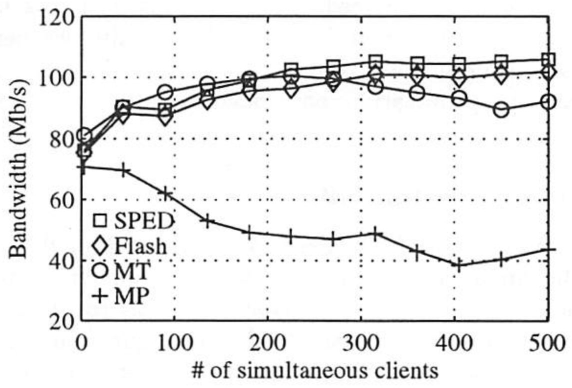


Figure 12: Adding clients - The low per-client overheads of the MT, SPED and AMPED models cause stable performance when adding clients. Multiple application-level caches and per-process overheads cause the MP model's performance to drop.

Web server benchmarking in a LAN environment fails to evaluate an important aspect of real Web workloads, namely that fact that clients contact the server through a wide-area network. The limited bandwidth and packet losses of a WAN increase the average HTTP connection duration, when compared to LAN environment. As a result, at a given throughput in requests/second, a real server handles a significantly larger number of concurrent connections than a server tested under LAN conditions [24].

The number of concurrent connections can have a significant impact on server performance [4]. Our next experiment measures the impact of the number of concurrent HTTP connections on our various servers. Persistent connections were used to simulate the effect of long-lasting WAN connections in a LAN-based testbed. We replay the ECE logs with a 90MB data set size to expose the performance effects of a limited file cache size. In Figure 12 we see the performance under Solaris as the number of number of simultaneous clients is increased.

The SPED, AMPED and MT servers display an initial rise in performance as the number of concurrent connections increases. This increase is likely due to the added concurrency and various aggregation effects. For instance, a large number of connections increases the average number of completed I/O events reported in each `select` system call, amortizing the overhead of this operation over a larger number of I/O events.

As the number of concurrent connections exceeds 200, the performance of SPED and AMPED flattens while the MT server suffers a gradual decline in performance. This decline is related to the per-thread switching and space overhead of the MT architecture. The MP model suffers from additional per-process overhead, which results in a significant decline in performance as the number of concurrent connections increases.

7 Related Work

James Hu et al. [17] perform an analysis of Web server optimizations. They consider two different architectures, the multi-threaded architecture and one that employs a pool of threads, and evaluate their performance on UNIX systems as well as Windows NT using the WebStone benchmark.

Various researchers have analyzed the processing costs of the different steps of HTTP request serving and have proposed improvements. Nahum et al. [25] compare existing high-performance approaches with new socket APIs and evaluate their work on both single-file tests and other benchmarks. Yiming Hu et al. [18] extensively analyze an earlier version of Apache and implement a number of optimizations, improving performance especially for smaller requests. Yates et al. [31] measure the demands a server places on the operating system for various workloads types and service rates. Banga et al. [5] examine operating system support for event-driven servers and propose new APIs to remove bottlenecks observed with large numbers of concurrent connections.

The Flash server and its AMPED architecture bear some resemblance to Thoth [9], a portable operating system and environment built using "multi-process structuring." This model of programming uses groups of processes called "teams" which cooperate by passing messages to indicate activity. Parallelism and asynchronous operation can be handled by having one process synchronously wait for an activity and then communicate its occurrence to an event-driven server. In this model, Flash's disk helper processes can be seen as waiting for asynchronous events (completion of a disk access) and relaying that information to the main server process.

The Harvest/Squid project [8] also uses the model of an event-driven server combined with helper processes waiting on slow actions. In that case, the server keeps its own DNS cache and uses a set of "dnsserver" processes to perform calls to the `gethostbyname()` library routine. Since the DNS lookup can cause the library routine to block, only the dnsserver process is affected. Whereas Flash uses the helper mechanism for blocking disk accesses, Harvest attempts to use the `select()` call to perform non-blocking file accesses. As explained earlier, most UNIX systems do not support this use of `select()` and falsely indicate that the disk access will not block. Harvest also attempts to reduce the number of disk metadata operations.

Given the impact of disk accesses on Web servers, new caching policies have been proposed in other work. Arlitt et al. [2] propose new caching policies by analyzing server access logs and looking for similarities across servers. Cao et al. [7] introduce the Greedy DualSize caching policy which uses both access frequency and file size in making cache replacement decisions. Other work has also analyzed various aspects of Web server workloads [11, 23].

Data copying within the operating system is a significant cost when processing large files, and several approaches have been proposed to alleviate the problem. Thadani et al. [30] introduce a new API to read and send memory-mapped files without copying. IO-Lite [29] extends the fbufs [14] model to integrate filesystem, networking, interprocess communication, and application-level buffers using a set of uniform interfaces. Engler et al. [20] use low-level interaction between the Cheetah Web server and their exokernel to eliminate copying and streamline small-request handling. The Lava project uses similar techniques in a microkernel environment [22].

Other approaches for increasing Web server performance employ multiple machines. In this area, some work has focused on using multiple server nodes in parallel [6, 10, 13, 16, 19, 28], or sharing memory across machines [12, 15, 21].

8 Conclusion

This paper presents a new portable high-performance Web server architecture, called asymmetric multi-process event-driven (AMPED), and describes an implementation of this architecture, the Flash Web server. Flash nearly matches the performance of SPED servers on cached workloads while simultaneously matching or exceeding the performance of MP and MT servers on disk-intensive workloads. Moreover, Flash uses only standard APIs available in modern operating systems and is therefore easily portable.

We present results of experiments to evaluate the impact of a Web server's concurrency architecture on its performance. For this purpose, various server architectures were implemented from the same code base. Results show that Flash with its AMPED architecture can nearly match or exceed the performance of other architectures across a wide range of realistic workloads.

Results also show that the Flash server's performance exceeds that of the Zeus Web server by up to 30%, and it exceeds the performance of Apache by up to 50% on real workloads. Finally, we perform experiments to show the contribution of the various optimizations embedded in Flash on its performance.

Acknowledgments

We are grateful to Erich Nahum, Jeff Mogul, and the anonymous reviewers, whose comments have helped to improve this paper. Thanks to Michael Pearlman for our Solaris testbed configuration. Special thanks to Zeus Technology for use of their server software and Damian Reeves for feedback and technical assistance with it. Thanks to Jef Poskanzer for the thttpd web server, from which Flash derives some infrastructure. This work was supported in part by NSF Grants CCR-9803673, CCR-9503098, MIP-9521386, by Texas TATP Grant 003604, and by an IBM Partnership Award.

References

[1] Apache. http://www.apache.org

[2] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference*, pages 126–137, Philadelphia, PA, Apr. 1996.

[3] G. Banga and P. Druschel. Measuring the capacity of a Web server. In *Proceedings of*

the USENIX Symposium on Internet Technologies and Systems (USITS), Monterey, CA, Dec. 1997.

[4] G. Banga and P. Druschel. Measuring the capacity of a Web server under realistic loads. *World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation)*, 1999. To appear.

[5] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. 3rd USENIX Symp. on Operating Systems Design and Implementation*, Feb. 1999.

[6] T. Brisco. DNS Support for Load Balancing. RFC 1794, Apr. 1995.

[7] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[8] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Technical Conference*, Jan. 1996.

[9] D. R. Cheriton. *The Thoth System: Multi-Process Structuring and Portability.* Elsevier Science Publishing Co,. Inc, 1982.

[10] Cisco Systems Inc. LocalDirector. http://www.cisco.com

[11] M. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *Proceedings of the ACM SIGMETRICS '96 Conference*, pages 160–169, Philadelphia, PA, Apr. 1996.

[12] M. Dahlin, R. Yang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. USENIX Symp. on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994.

[13] O. P. Damani, P.-Y. E. Chung, Y. Huang, C. Kintala, and Y.-M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems*, 29:1019–1027, 1997.

[14] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 189–202, Dec. 1993.

[15] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995.

[16] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[17] J. C. Hu, I. Pyrali, and D. C. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Proceedings of the 2nd Global Internet Conference*, Phoenix, AZ, Nov. 1997.

[18] Y. Hu, A. Nanda, and Q. Yang. Measurement, analysis and performance improvement of the Apache web server. In *Proceedings of the 18th IEEE International Performance, Computing and Communications Conference (IPCCC'99)*, February 1999.

[19] IBM Corporation. IBM eNetwork dispatcher. http://www.software.ibm.com/network/dispatcher

[20] M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. Server Operating Systems. In *Proceedings of the 1996 ACM SIGOPS European Workshop*, pages 141–148, Connemara, Ireland, Sept. 1996.

[21] H. Levy, G. Voelker, A. Karlin, E. Anderson, and T. Kimbrel. Implementing Cooperative Prefetching and Caching in a Globally-Managed Memory System. In *Proceedings of the ACM SIGMETRICS '98 Conference*, Madison, WI, June 1998.

[22] J. Liedtke, V. Panteleenko, T. Jaeger, and N. Islam. High-performance caching with the Lava hit-server. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, LA, June 1998.

[23] S. Manley and M. Seltzer. Web Facts and Fantasy. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 125–134, Monterey, CA, Dec. 1997.

[24] J. C. Mogul. Network behavior of a busy web server and its clients. Technical Report WRL 95/5, DEC Western Research Laboratory, Palo Alto, CA, 1995.

[25] E. Nahum, T. Barzilai, and D. Kandlur. Performance Issues in WWW Servers. submitted for publication.

[26] National Center for Supercomputing Applications. Common Gateway Interface. http://hoohoo.ncsa.uiuc.edu/cgi

[27] Open Market, Inc. FastCGI specification. http://www.fastcgi.com

[28] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998. ACM.

[29] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[30] M. N. Thadani and Y. A. Khalidi. An efficient zero-copy I/O framework for UNIX. Technical Report SMLI TR-95-39, Sun Microsystems Laboratories, Inc., May 1995.

[31] D. Yates, V. Almeida, and J. Almeida. On the interaction between an operating system and Web server. Technical Report TR-97-012, Boston University, CS Dept., Boston MA, 1997.

[32] Zeus Technology Limited. Zeus Web Server. http://www.zeus.co.uk

NewsCache – A High Performance Cache Implementation for Usenet News*

Thomas Gschwind Manfred Hauswirth
{tom,M.Hauswirth}@infosys.tuwien.ac.at
Distributed Systems Group
Technische Universität Wien
Argentinierstraße 8/E1841
A-1040 Wien, Austria, Europe

Abstract

Usenet News is reaching its limits as current traffic strains the available infrastructure. News data volume increases steadily and competition with other Internet services has intensified. Consequently bandwidth requirements are often beyond that provided by typical links and the processing power needed exceeds a single system's capabilities. A rapidly growing number of users, especially attracted by WWW, overloads communication links and makes bandwidth a scarce resource. While an elaborate caching infrastructure was adopted for the WWW, Usenet News still uses most of its originally defined infrastructure. Caching techniques have not yet been adopted on a large scale. We believe that this is due to the lack of efficient cache implementations. In this paper we present a high performance cache server for Usenet News that helps to conserve network bandwidth, computing power, and disk storage and is compatible with the current infrastructure and standards. After a thorough comparison of existing news database formats and replacement strategies we designed and implemented NEWSCACHE to remedy Usenet News bottlenecks. We present an empirical comparison of different cache replacement strategies as well as an evaluation of the use of NEWSCACHE as a news server.

1 Introduction

Usenet News provides a global distributed blackboard on top of other networks. It consists of a set of hierarchical newsgroups which are dedicated to specific topics. *Articles* or *messages* are submitted (*posted*) to one or more newsgroups and are replicated to all Usenet sites holding one of the newsgroups the article was posted to [1]. The newsgroups that are stored on a news server and thus provided to its clients are defined by the news server's administrator.

The world-wide set of cooperating news servers makes up the distribution infrastructure of the News system. Articles are distributed among news servers using the Network News Transfer Protocol (NNTP) which is defined in RFC977 [2]. In recent years several extensions have been applied to NNTP. These are currently available as an Internet draft [3] which will supersede RFC977 within the next few months. The format of Usenet messages is defined in [4].

News readers provide the user interface for reading News and interact with their news server using the Network News Reader Protocol (NNRP). NNRP is actually the subset of NNTP that is used by news readers. Since the news reader stores data specific to each news server, such as article numbers to keep track of read articles, the user must always connect to the same news server to get a consistent view.

The number of newsgroups (currently about 55000[1])and users is growing steadily. At the infrastructure level this implies growing amounts of data that need to be distributed and pushes existing News infrastructure to its limits [5, 6]. Section 2 gives an introduction to the current News infrastructure, its problems—which are mainly caused by the lack of scalability due to News's n copy semantics that causes high bandwidth and resource consumptions—and possible solution strategies.

A news server maintains a set of databases and log files to store and monitor the news spool which are central to the performance of the system as a whole. These databases are explained in Section 3 along with a comparison of the organization of these databases as implemented in various well-known news servers and NEWSCACHE itself.

*This work was supported in part by a grant from the Hewlett-Packard European Internet Initiative.

[1]This figure is taken from `news.tuwien.ac.at` and may vary according to the newsfeed available on a specific news server.

In Section 4 we present the design and implementation of NEWSCACHE. We explain the replacement strategy used for NEWSCACHE based on an analysis of various replacement strategies that can be used for a caching news server. Besides caching, our NEWSCACHE provides additional and new features which are described in Section 5. Section 6 evaluates our implementation and Section 7 gives an outlook on future work.

Related work is considered in Section 8 followed by our conclusions in Section 9.

2 News, its Problems, and a Solution

When a user posts an article to a newsgroup, the news reader transfers it to its news server via NNRP. Then the article is distributed among the news servers using the Network News Transfer Protocol (NNTP) [2, 3]. The news server keeps a copy of the article and forwards (*feeds*) it to its "neighboring" news servers that also hold the newsgroup that the article has been posted to. Those servers in turn forward it to their neighbors until all news servers that hold the newsgroup have received a copy of the article.

No restrictions exist on the topology with which articles are distributed among the news servers. To prevent duplicate delivery of the article two strategies are applied: each article carries a unique message identifier that allows a news server to identify whether it has already seen an article; additionally a news server adds its own name to the *path* header of every article it receives. This allows a news server to identify which news servers already have seen this article and feeding to those is not necessary [7].

This simple infrastructure of News has provided flexible and reliable service over the past years. The analysis of logfiles of large news servers, however, shows a doubling of article numbers nearly every 18 months by occasional bursts of growth [5]. In the current infrastructure these figures can easily be mapped onto network bandwidth requirements which are likely to grow at a similar rate.

A main scalability problem stems from News's *n* copy distribution semantics. As described above each article is copied to every news server holding the relevant newsgroup(s). Since a high percentage of the articles will not be read by anyone, copying all articles is highly redundant for a leaf node news server. Measurements at our university's news server have shown that only 20% of all available newsgroups are actually read [8].

Currently a typical newsfeed requires the transfer of about 3–5GB of article data per day [6]. For a typical site connected to the Internet via a T1 link (1.5 MBit/s) News's bandwidth requirements account for up to 35% of the total available bandwidth. This is the lower bound to guarantee news distribution without generating a backlog. A backlog might not be recoverable due to limited bandwidth and computing resources. If this occurs, the news service has to be decreased in terms of fewer available newsgroups and randomly unavailable articles.

Another problem that has to be accounted for is the I/O load on the news server caused by the news traffic [9]. To illustrate this, consider our university's hardware requirements for News: `news.tuwien.ac.at` is an UltraSPARC 2 system with 2 168MHz CPUs, 512MB main memory and uses highly-optimized NFS server hardware for the spool-directory, i.e. a 100GB RAID4 file server with a file system (WAFL) highly optimized for directory accesses and big directories, connected via 100MBit FDDI to the UltraSPARC. Even with this machine the number of newsgroups provided had to be cut down from about 45000 to 6000 newsgroups. Users can request additional newsgroups (from a total of 55000 groups at the moment) via a WWW page [8].

News servers are responsible for both article distribution and providing news service to their clients. Additional hardware requirements may be imposed by a high number of news clients. An architecture that separates these two functionalities into a distribution backbone and an access infrastructure would provide higher flexibility and scalability. This separation can be implemented by administrative and management measures [6] or by a new News infrastructure. In [10] we compared infrastructures for News and came to the following conclusions:

- The access infrastructure should be separated from the distribution infrastructure using cache servers. This makes it possible to provide a virtual full feed over a T1 link (1.5MBit/s) or a slower link.
- Leaf node news servers that are not part of the news distribution infrastructure can be replaced by cache servers.
- Servers with a full spool should form a distribution backbone where news articles are exchanged using multicast. Distribution of News via multicast is discussed in [11].
- If all clients were able to retrieve articles from several news servers, it would not be necessary for every news server to store all newsgroups that users might want to read.

NEWSCACHE is a cache server implementation intended to access the News infrastructure. Since it only uses NNRP, it fits seamlessly into the existing infrastructure without requiring modifications to existing software. Each requested article is stored locally by NEWSCACHE to satisfy successive requests without having to contact the news server again. This eliminates additional transfers, thus conserving bandwidth; decreases load on the news server; and reduces disk space requirements since only articles that actually are accessed need to be stored. Given n_i is the number of accesses and sz_i is the size of article i the reduction in bytes can be approximated as

$$Reduction = \sum (n_i - 1) * sz_i \quad \text{[Bytes]}$$

This formula represents only the upper bound. An article that has been replaced by another will have to be requested again, if it is again by a user.

Caching is ideally applicable for News because of the lack of updates: articles do not change over time; they can only be added to a newsgroup or expire. This simplifies the application of caching considerably.

3 News Database

Each news server maintains a set of databases that store articles and newsgroups as well as meta-information. In the following sections we will explain the purpose of each database, how they are implemented by various news servers, and the improvements NEWSCACHE offers for each database. We looked at important and widespread news server implementations, such as the NNTP reference implementation [12], c-news [13] which is historically important and still has a large number of users, and INN [7] [14] which is the news server with the most installations nowadays. We have also included NNTPCACHE's organization of the news database [15]—another cache server for News that was developed in parallel with NEWSCACHE.

3.1 Active Database

The *active database* stores a list of all the newsgroups available on the news server along with the number of the first article (low watermark), the number of the last article (high watermark), a posting flag, which indicates the type of the newsgroup, and the creation times of a newsgroup. Articles are numbered sequentially within a newsgroup. Articles posted to different newsgroups, will have several article numbers (one for each newsgroup). The article number is site-specific, depending on the arrival order and must never be reused within a newsgroup.

For the news server, the active database is necessary to calculate the article number(s) of newly arriving articles within their newsgroup(s). For the news reader, it provides an overview of the newsgroups available from the news server, the date the newsgroup has been created, and allows the news reader to estimate the total number of articles ($total = hi\_watermark - lo\_watermark + 1$) and the number of unread articles ($articles\_unread = total - articles\_read$) within each newsgroup.

Traditionally the active database is stored in two files (`active` and `active.times`). One listing the watermarks and the moderation status and the other one giving the creation time of the newsgroups (c-news, INN, NNTP reference implementation).

NEWSCACHE, however, stores all this information in one memory mapped database including the number of articles available in each newsgroup and a timestamp when a newsgroup has been requested the last time from the news server.

An advantage of storing the number of articles within the active database is that articles need not be counted whenever a newsgroup is selected while still being able to provide an accurate count of the articles present in the newsgroup.

NEWSCACHE uses timestamps to synchronize with its upstream news server. A configurable threshold value controls the frequency of cache updates (consistency checks) from the server. This is a trade-off between cache coherence and bandwidth/connections to the server. A time interval of 0 for the update period of the database provides a fully coherent view of the database at the cost of increased connections (bandwidth). A bigger value means that articles will be available to cache users with a slight delay.

As far as we could find out, NNTPCACHE organizes its active database similar to NEWSCACHE. All the information is kept in one memory mapped file.

3.2 Article and Newsgroup Database

The article and newsgroup database stores all the news articles and maps the articles to the newsgroups which they have been posted to. Traditionally the newsgroup hierarchy is mapped onto a directory hierarchy and the articles are stored in separate files in their newsgroup directory. If an article is posted to several groups hard links are used to prevent multiple storing of an article (c-news, NNTP reference implementation, NNTPCACHE).

INN uses the same format but is able to use soft links which allows distribution of the News spool over multiple file systems.

However, this approach has some drawbacks due to the fact that the average size of an article is rather small (about 2.5KB) [5, 6].

- On filesystems using a limited number of files per filesystem one can run out of file entries (inodes), despite enough disk space is available.
- On filesystems with block sizes of 1 to 4KB up to 100% of the real data volume can be wasted [5].
- Newsgroups with heavy traffic tend to be bottlenecks due to a linear lookup of files in the directory structure.

To overcome the problems of this storage format, the current version of INN supports two alternatives: the *timehash* format is similar to the traditional format, but articles are divided into subdirectories based on the arrival timestamp. The *cnfs* format uses pre-configured buffer files of a configurable size for every newsgroup. Upon reaching the end of the buffer file, new articles are stored again from the beginning of the file. However, the disadvantage of this approach is that the article retention time cannot be controlled since articles get overwritten automatically when the buffer is full. The installation manual recommends to use the traditional format. Only if a full feed has to be maintained, *cnfs* is recommended. [14]

Even though *cnfs* seems to be the best choice from a performance point of view, it cannot be used for a cache server because the size of the buffer files is fixed and available file system space cannot be flexibly reallocated between different newsgroups. Another disadvantage is that the article retention time cannot be controlled which is important for caching.

NEWSCACHE in contrast uses a database for each newsgroup that stores articles with a size of less than 16KB. Larger articles are stored in the filesystem to keep the newsgroup database small and to improve the database's caching behavior. The article size threshold is user-configurable and should be chosen in a way to utilize the filesystem's block allocation as efficient as possible (most filesystems allocate disk space in chunks of 2^nKB, where n is constant).

The advantage of our approach is that it does not suffer from a limited number of files and the filesystem lookup is required for big articles (articles bigger than 16KB) only. However, newsgroups with many large articles still depend on the filesystem's organization.

Since an article's unique message identifiers and the per newsgroup article numbers must not be reused for other articles, the article itself has a virtually infinite lifetime. Thus the article itself cannot be changed and thus no coherency control messages are necessary which makes the caching of articles efficient and easy.

When an article is submitted to or expired from a newsgroup the low and high watermarks of the newsgroup will change accordingly. This can be easily identified using NNTP's `group` command which selects a newsgroup and reports the newsgroup's watermarks.

However, articles can be added to or removed from a newsgroup not only by submission or expiration but also via cancellation messages or by article reinstatement [3, 2]. While the former can be done by every Usenet user, the latter can only be done by the news server's administrator. Failing to detect article cancellation is only a minor flaw but failing to detect article reinstatement has the effect that the user might miss these articles. Reinstatement of an article conforms to RFC977 [2] but since it occurs seldomly only few news readers handle it correctly. Most news readers mark articles not available on the news server as read which might not be the case (i.e., tin, xrn, slrn, MS Outlook Express).

NEWSCACHE uses the `listgroup` command to obtain a list of all valid article numbers within the current newsgroup. Based on this list NEWSCACHE can identify a canceled article (the article exists in the cache but not on the server) or a reinstated article (the article exists on the server but not in the cache).

3.3 Overview Database

The *overview database* stores a short summary for each article. Using the overview database, a news reader can provide a faster overview of the articles available in a newsgroup.

INN stores the overview database for each newsgroup as a single plain text file. Whenever an article is posted to a newsgroup, its overview record is appended to this file. If an article gets deleted or is expired in the newsgroup, the whole file has to be rewritten.

NNTPCACHE takes the same approach except that the overview records for one group are split up into several files with 512 overview records per file. This is necessary since otherwise the whole overview database would have to be rewritten when a new record has to be inserted at the beginning of the file. This occurs frequently because NNTPCACHE has no control of when a client requests an overview record. Some news readers even request the overview records in reverse order, to be able to present the newest articles first to the user while older records are being retrieved (e.g., Netscape). NNTPCACHE generates overview records on the fly if an article is not available and stores them in the overview database.

NEWSCACHE always generates overview records for articles stored within the newsgroup's memory mapped database on the fly. Overview records for other articles (articles that have not been cached yet or articles bigger than 16KB) are also stored in the newsgroup's database. This reduces the disk space necessary for NEWSCACHE by approximately 20%. Additionally, the overview database has its own memory allocation methods based on memory mapped files to allow changes without the need to rewrite the entire file.

A record in the overview database represents a subset of the corresponding article. Thus the same requirements as for the article database account for the overview database. When a news reader requests an overview record for a newsgroup, NEWSCACHE immediately prefetches all entries not yet cached since it is the most frequently used information. Most news readers will retrieve the overview database of the whole newsgroup whenever the newsgroup is selected by the user (e.g., tin, xrn)

3.4 History Database

The history database stores meta information about articles and newsgroups. It stores the arrival time and expiration of an article along with its message identifier. This allows the news server to identify whether an article is offered again by another news server but has already been expired on the local server. It stores the creation and deletion time of newsgroups. The creation time is necessary to be able to inform news readers of newly created newsgroups.

Usually, the history database is managed like a log file (c-news, INN), where new entries are appended at the end of the file. NEWSCACHE stores the creation times of the newsgroups within its active database. Storing the articles' message identifier is not necessary because NEWSCACHE does not participate in the distribution of articles between news servers. It provides an exact image of its upstream news server, thus freeing NEWSCACHE from having to identify duplicate articles.

4 Implementation of NewsCache

NEWSCACHE has been designed to be easily extendible and reusable. For this purpose we implemented a news server class library that provides an interface to different kinds of news databases. The hierarchy is shown in Figure 1 using the notation presented in [16].

The abstract news server class (NServer) defines the interface that has to be implemented by all the news server classes and provides a factory method (`getgroup()`) that lets the user create the news server's news database.

The local server class (LServer) implements an interface to a local news database. This class can serve as a base class for the implementation of a news server trimmed to the user's requirements. The remote server class (RServer) implements an interface to access a database provided by a news server. It also allows for multiplexing between different news servers on a per newsgroup basis. The RServer class can be used as the communication interface to a news server for a new news reader. The cache server class (CServer) inherits the functionality from the LServer and the RServer classes. As a result, CServer provides an interface to the database of a news server with the ability to cache messages in the news database provided by the LServer class.

Each news database is defined by its own interface (inherited from an abstract class). The access to the databases is controlled by the news server classes which means that a reference to the newsgroup databases has to be requested from the server component before a user (for NEWSCACHE this is the `nnrpd` class that handles client requests) of the library can access the newsgroup.

For the databases used by the LServer and CServer classes, we implemented a *Non Volatile Container* class library. Instead of using the heap for memory allocation, the class library provides its own functions for allocating and freeing memory by using memory mapped files. Using this memory allocation model we implemented a set of containers. NVcontainer provides all the methods necessary for allocating and freeing memory from the mapped file. This functionality is inherited by all the subclasses of NVcontainer. At the moment we provide a list container (NVList), an externally linked hash table (NVHash), and an array (NVArray). Figure 2 shows the hierarchy of this library.

Using memory mapped files has several advantages. The size of the database is not limited by the available virtual memory. A container's content need not be written to or read from the disk explicitly. If the operating system caches file accesses, this approach has no performance penalty over the use of main memory. No swap space is consumed by this container class, since the cached data will be written back to the file instead of to the operating system's swap space. If the database should be shared by a set of processes, changes are visible to other processes immediately and no shared memory has to be

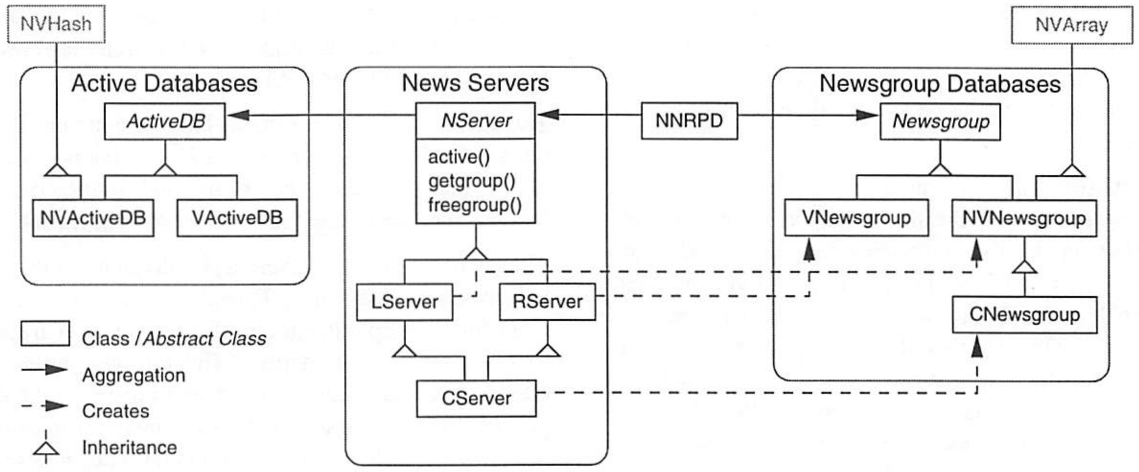


Figure 1: NEWSCACHE's class hierarchy

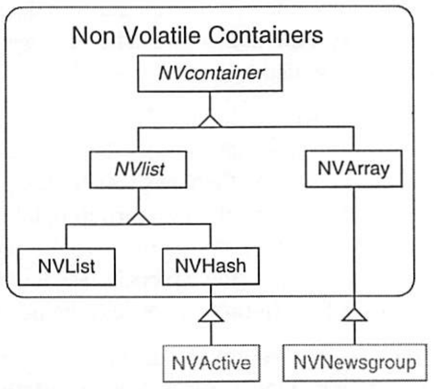


Figure 2: Inheritance hierarchy of the *Non Volatile Container Class* library

allocated. However, memory mapped files also have a drawback: whenever more memory is necessary than available, the file has to be resized and remapped to a possibly different memory location. To reduce the performance penalty of this operation we allocate space in bigger chunks (64KB at the moment).

For our implementation of the active database, we use the NVHash container. We use a externally linked hash table because it is not as complex as a balanced tree and more efficient as long as the number of elements is predictable. Since the number of elements can be estimated *a priori* for the next few years this is not a big issue. Currently about 55000 newsgroups exist and up to now this number doubles every 18 months. Thus a hashtable with about 20000 entries will be sufficient for the next two years (about 120000 newsgroups divided by 20000 entries gives between 1 and 6 comparisons per lookup). Then the size of the hash table should be reconfigured which takes only a few seconds. Our implementation stores all information of the active database within the same database.

For the newsgroup database, each newsgroup uses its own database to store news articles. Each database is stored within its own directory. The name of the directory is derived from the name of the newsgroup (similar to INN).

Compared to other implementations, we provide a sophisticated newsgroup database. Only big articles (articles bigger than 16KB at the moment) are stored using a separate file. All other information, be it a small article or an overview record, are stored in the same database. This reduces the number of files and keeps related information together and thus improves caching behavior. To further reduce disk-space requirements we went one step beyond and store overview records only for articles that are stored externally or for articles where only the overview record has been requested. This performance penalty (about 20% as we will show in Section 6) is outweighed by the disk space reduction, since the overview database is up to 20% of the size of the news database.

4.1 Choice of Replacement Strategy

A key determinant of the performance of cache systems is the replacement strategy. We have compared the replacement strategies applicable to our domain in terms of the resulting network bandwidth and in their hit rates respectively. Where meaningful, the replacement strategies were compared on a per article basis and on a per newsgroup basis (the smallest unit to be removed is an article or a newsgroup respectively). For the per newsgroup replacement strategies it is important to note that articles will also be removed when they are expired on the upstream news server. Otherwise the newsgroup will grow infinitely. A comparison of the strategies is depicted in Figure 3 in terms of consumed network bandwidth and in Figure 4 in terms of their hit rates. The replacement strategies have been simulated using access patterns obtained from NEWSCACHE's log files (logged over a 10 days period).

It is interesting to note that a better hit rate does not necessarily imply less network bandwidth consumption. For instance *biggest article first* has nearly always a better hit rate than *least frequently used* but in some situations transfers more bytes from its upstream news server.

What we did not expect was that LRU on a per newsgroup basis (LRUG) performs better than LRU on a per article basis (LRUA). We assumed that LRUA has a finer granularity and thus will perform better. Our interpretation is that the newsgroup should be seen as a unit and that an article's access probability often can be estimated better by looking at all the articles in the group than by just looking at one article. Sometimes this generalization is not true and the hit rate can degrade even though the spool size is increased.

We considered the following replacement strategies:

BAF removes the biggest articles first, thus favoring newsgroups with small articles. This strategy assumes that only a few users read binary newsgroups[2] and that those users should be penalized. This strategy is good when a good hit-rate for *ordinary* News users (users not reading binary newsgroups) should be provided.
However, if many people are reading binary newsgroups, as in our case, the hit rate will be poor. Thus if the major concern is to reduce the required network bandwidth BAF does not perform well.

LFUA/LFUG removes the least frequently used article (LFUA) or least frequently newsgroup (LFUG) first. As expected LFUA performs poorly. It favors older articles that have been read more frequently and thus have already been read by most users.
LFUG performs considerably better since it takes the overall interest in the newsgroup into account. The only problem is that LFU cannot adopt to new requirements quickly. This seems to be the main reason why it performs bad on small article spools and why it oscillates so heavily on small to average sized spool sizes.

LRUA/LRUG removes least recently used articles (LRUA) or newsgroups (LRUG) first. This strategy assumes that items that have not been accessed for a long time are no longer of interest. LRUA can adopt faster to changing requirements than LFUA because it does not take old accesses into account. Thus, as we expected the least recently used strategy performs better than LFUA.

LETF removes articles with the least expiration time first. The drawback of this approach is that it treats all groups the same and does not take the article's access patterns into account. However, it is better then BAF or LFUA since it takes into account that older articles are less interesting to Usenet users and thus are less likely to be accessed in the future.

Initially we used a per newsgroup replacement strategy because it was the easiest way to implement article replacement and imposed the smallest CPU load. Then we tested a hybrid replacement strategy (LRUA and LRUG mix) because we assumed that a per article replacement strategy would perform better. This did not prove true so we have gone back to LRUG.

5 New and Additional Features

NEWSCACHE's design rationale is compatibility, scalability, and extendibility. Compatibility with the existing infrastructure and scalability issues already have been addressed in previous sections. By extendibility we mean that we have included new functionality into NEWSCACHE and its design eases the addition of new features.

Many news readers can interact with only one news server, thus tying the user to the server's newsgroup selection. NEWSCACHE can remedy this situation by its *transparent multiplexing* functionality: it can simultaneously cooperate with a set of news servers and combines them into one virtual news server for its clients. This feature can also be utilized for infrastructural improvements: *newsgroups can be partitioned* among a set of news servers and access is done via NEWSCACHE. Done at an appropriate organizational scale, this can decrease network bandwidth consumption and I/O load on the news servers, while users still have access to all newsgroups.

Additionally the multiplexing feature can be used for the provision of *local newsgroups*. NEWSCACHE can be setup to multiplex between the newsfeed and a local news server that only holds groups of local scope.

[2]Newsgroups that mainly distribute programs or other big data like pictures are usually called binary newsgroups—some people think that those newsgroups should be banned from Usenet.

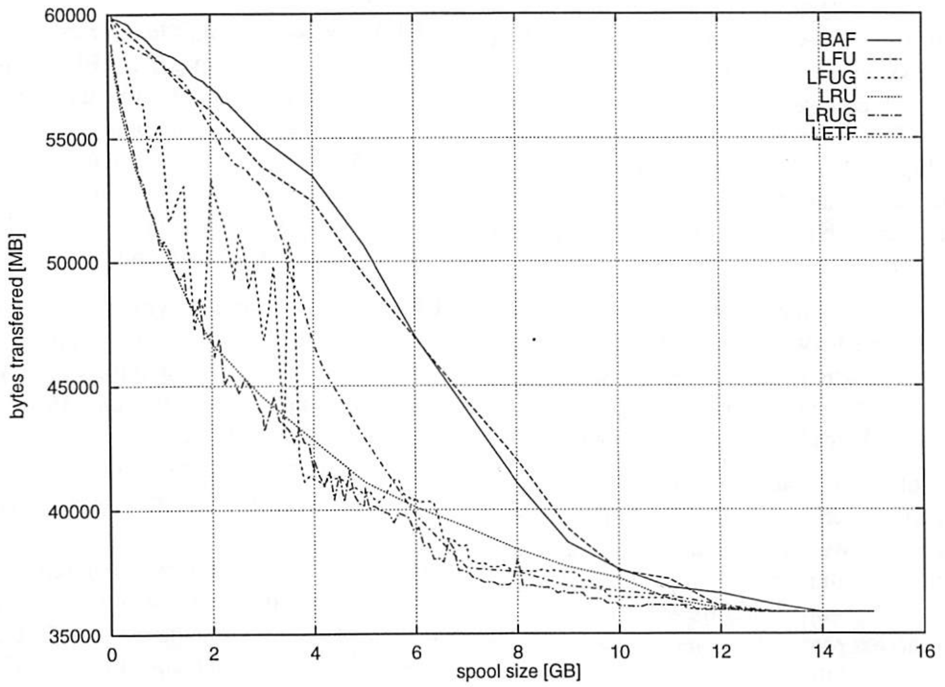


Figure 3: Bandwidth based on replacement strategies with varying spool sizes

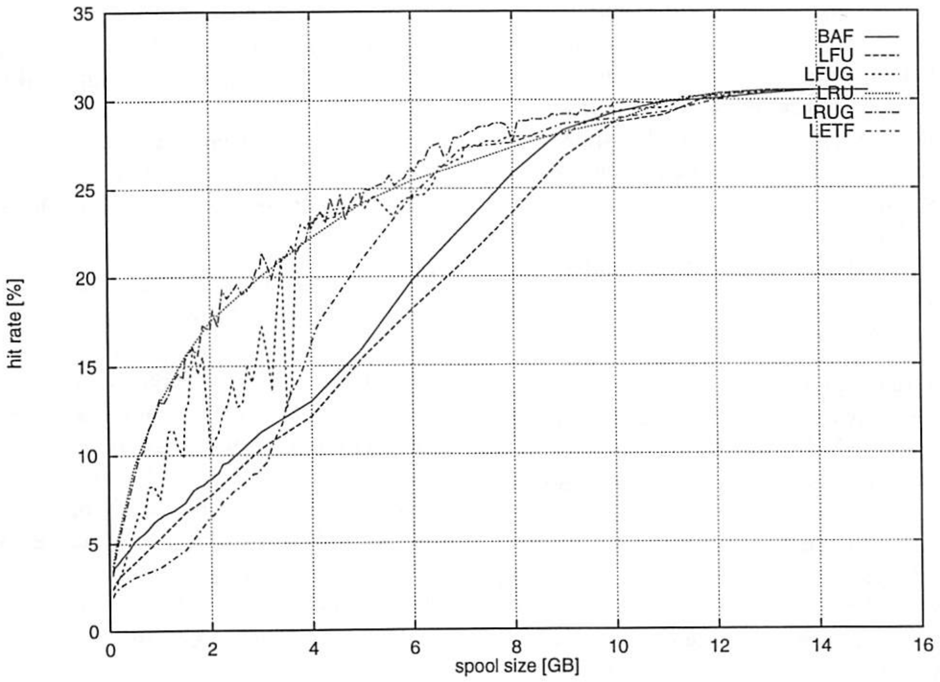


Figure 4: Hit rates of replacement strategies with varying spool sizes

Since this functionality requires setting up a local news server whose functionalities are not exploited in this setting, we plan to include a light-weight news server into NEWSCACHE for this purpose.

NEWSCACHE also supports access to a virtual full *newsfeed over limited bandwidth network connections.* This functionality can be combined with a *prefetching* strategy: a set of newsgroups can be retrieved in off-hours when the network is less loaded and then will be accessible much faster during network peak time. This functionality can also be bestowed to provide *offline news reading*. By using the prefetching functionality newsgroups are in the cache and can then be read offline. Postings submitted during offline operation are queued for later submission. However, during offline operation, NEWSCACHE cannot retrieve articles that have not been cached. Thus, those articles cannot be presented to the news reader. Unfortunately, many news readers assume then that the article has been deleted and marks it as read and will never present it to the user even if the article is reinstated by NEWSCACHE when a connection to the upstream news server is re-established.

6 Evaluation

After developing and implementing NEWSCACHE we evaluated the success of our design decisions and the validity of our assumptions about the weaknesses of the other server's database organization. We performed the evaluation in several experimental setups. In the following we will present our results that validate our design decisions. As we will show, however, the results also indicate that a small performance increase might still be possible.

Table 1 shows the performance of NEWSCACHE's News database compared to NNTPCACHE's and INN-2.0's performance. Our test machine was a Pentium 200MHz, 64MB main memory, and 3GB IDE hard disk running Linux 2.0.35. The cache servers and INN were running on the same machine since we wanted to know the minimal latency involved when retrieving News via each cache server. It shows that a miss imposes only little overhead and in the case of a hit NEWSCACHE performs better than its competitors except for the retrieval of the active and overview databases where it performs as good as NNTPCACHE.

Table 2 compares the delay of hits, misses, and direct connections in a realistic setup. Clients are located on the same LAN as NEWSCACHE. The LAN is connected to the news server over a small bandwidth link (i.e., 56kBit modem line). Table 3 includes a rough calculation of the performance advantage that will be experienced by users that use NEWSCACHE by combining the figures in found Table 2 and 3.

Description	NEWSCACHE		INN
	Hit	Miss	
Active database retrieval	10.1s	120s	110s
Retrieving 1000 articles	27.5s	523s	471s
Overview database of 5 groups[a]	40.3s (41.3s)	373s	331s
Selection of 350 newsgroups	0.7s	43.6s	43.3s

[a] The second figure indicates the access time of the overview database when it is generated on the fly.

Table 2: Accessing News over a slow (56kBit) link with NEWSCACHE installed locally.

Another interesting thing to note is that the delay of some operations are masked out compared to Table 1 (i.e., on the fly generation of the overview database) due to the fact that the client and NEWSCACHE are running on different machines. Other operations do not profit from this. We see this as an indication that retrieving the active database from the server and sending it to the client can be optimized by interleaving those operations in a better way.

For a cache not only good performance values but also good hit rates are viable. Thus, we publicly announced the availability of NEWSCACHE at our university and asked people to use and test NEWSCACHE. The hit rates for this experiment are presented in Table 3. With more accesses to the cache we expect higher hit rates, especially when people have to use NEWSCACHE and cannot access the news server directly. Over a period of 10 days NEWSCACHE was accessed 6350 times from 309 hosts. In total 1314 different newsgroups were accessed among which the top 10 accounted for 59% of all accesses. So, reasonable locality in references to the newsgroups can be concluded.

	total	active	groups	ODB[a]	articles
requests	322934	941	40866	44159	205363
hits	86359	736	3326	10230	42204
	27%	78%	8%	23%	21%
perf-gain	18%[b]	69%	7%	10%	11%

[a] overview database.
[b] weighted average of group selection, active and overview database, and article retrieval. Other requests have not been considered for this figure.

Table 3: Hit Statistics (without Prefetching)

Description	NEWSCACHE Hit	NEWSCACHE Miss	NNTPCACHE Hit	NNTPCACHE Miss	INN
Active database retrieval	3.6s	7.0s	3.6s	8.4s	6.0s
Retrieving 1000 articles	19s	38s	24s	59s	20s
Overview database of 5 groups[a]	16.1s (20.8s)	111s	16.1s (128s)	125s	108s
Selection of 350 newsgroups	0.7s	21.8s	5.3s	22.8s	20.6s

[a]The second figure indicates the access time of the overview database when it is generated on the fly.

Table 1: Performance of NEWSCACHE

NEWSCACHE has been tested in combination with several news readers. Netscape works perfectly with NEWSCACHE, but we had to optimize the `group` command, since Netscape issues this command for each newsgroup to get a better estimation of the number of articles available within the newsgroups. Gnus, knews, MS Outlook Express, pine, slrn, tin, XRN also work in combination with NEWSCACHE. Other news readers have also been reported to work perfectly in combination with NEWSCACHE.

The following news servers have been tested in combination with NEWSCACHE: ANU News (VMS), INN, MS Internet Services, Netscape Collabra. No problem has been found so far.

7 Future Work

The active database changes whenever an article is submitted to a newsgroup or whenever a newsgroup is added or removed. Article submission and removal occurs much more frequently than newsgroup addition or removal. Unfortunately NNTP provides no command to check which entries of the active database have been modified since a given time. Only commands for retrieving the whole active database (`list active [wildmat]`) or only the part for newly added newsgroups (`newgroups`) exist. A *wildmat* expression can be applied to filter newsgroups based on their names. Thus whenever the active database needs to be updated the whole active database has to be requested (about 2MB).

Fortunately, the revised NNTP specification [3] is kept extendible enough to support custom extensions. Extensions supported by the news server can be retrieved using NNTP's `LIST EXTENSIONS` command. To overcome this problem we propose a slightly modified list command, `list active.modtime $time [wildmat]`, that provides a possibility to retrieve entries that have been changed since a given time. We will analyze the benefits of such a command and prepare a draft for an NNTP extension that defines this command.

The current implementation of our *Non Volatile Container Class* library is based on a code inheritance hierarchy. Even though this supports the changing of the type of container used during runtime it requires a virtual method call whenever the container is accessed. In future versions we will switch to a design based on templates similar to the design of STL [17].

At the moment the size of the hash table for the active database has to be specified when compiling NEWSCACHE. This should be a configuration option in NEWSCACHE's configuration file. However, this is not critical and adding this feature should be trivial.

We think that one of the key elements responsible for the good performance of NEWSCACHE is the *Non Volatile Container Class* library. In the future we will try to integrate this in INN and will evaluate whether INN can benefit from it.

We plan further analysis and experiments with cache replacement policies to find an optimal replacement policy for News. As our experiments have shown so far the application of such policies in the setting of News may yield unexpected results (see Section 4.1) and thus require further systematic study. News seems to differ from other cache application areas in a way that assumptions from other domains cannot be mapped 1:1 onto News.

Our measurements for News clients performance gains have been done indirectly via cache hit rates. While this provides a good approximation for the overall performance gains, it gives only a limited assessment of the performance gains for a single client. We want to use instrumented news readers to get such direct measurements. Additionally these results can be related to hit rates and other performance figures to get a better understanding of the runtime behavior and access profiles.

The selection process of the articles to be prefetched is another area of further investigation, i.e. how the information that is to be prefetched is chosen. Several scenarios seem to be useful besides prefetching based on the administrator's preferences: prefetch the newsgroups with the most user requests (in relation to the article sizes), thread based prefetching, etc.

8 Related Work

NNTPCACHE is the only system we found so far that is similar to NEWSCACHE, but no publications about it are available. The following statements are solely based on the documentation of NNTPCACHE's software distribution [15] and our tests with it.

NNTPCACHE offers censoring of articles, and forwarding of unknown commands. NEWSCACHE currently does not support these features but on the other hand offers functionality unknown to NNTPCACHE: prefetching, offline News reading, and `inetd` support.

An approach using a server with only a subset of the theoretically available newsgroups in combination with a web page where users can request the addition of new newsgroups available on the news server's news feed is explained in [8]. When a user requests a newsgroup via the web page it is supplied by the news server on the next day. The author explains that in average only 20% of all the theoretically available newsgroups are actively being read. However, this could be solved better using a cache server. This approach has the advantage that news users need not request new newsgroups via a web page since the cache server would offer all available newsgroups and the newsgroups would be available to the news user immediately.

Another approach where the News spool is partitioned among several computers is presented in [6]. While this cuts down the I/O load on each machine it does not target the network bandwidth consumption.

9 Conclusion

Despite the fact that consensus exists that caching must be applied to News in the presence of overloaded networks, only few approaches exist to attack this problem. These approaches alleviate the effects by applying management policies but do not attack the cause. NEWSCACHE, however, attacks this at the access infrastructure while still being compatible to existing news software.

NEWSCACHE can replace existing leaf node news servers thus reducing network bandwidth consumptions and reducing hardware requirements for the provision of Usenet News since only a fraction of the full News spool has to be stored while still providing a full feed to news users. NEWSCACHE can be used to speed up retrieval of News in environments where only a slow link to the news server exists. Another advantage of NEWSCACHE is that it offers news reading functionality only (postings are directly forwarded to the upstream news server) and needs not allocate resources and computing power for news distribution. This drastically cuts down on I/O load.

Even though NEWSCACHE is based on an object-oriented design whose design is not compromised by dirty performance hacks, it provides faster access to news articles than other state of the art news servers (i.e. INN). This is due to the fact that we did a thorough comparison of the design of the news database in various other news servers before implementing our own database.

The news database is based on memory mapped files using our own memory management. This approach allows us to manipulate persistent complex data structures as if they were stored on the heap. Other news servers such as INN might also benefit from this organization.

Another factor that has to be taken into account in the domain of caching is the replacement strategy used when the cache space fills up. We have compared different replacement strategies that can be employed when the spool size of the cache server fills up along with an analysis of the advantages of each. One surprising result was that when articles need to be replaced, it is better to remove older articles on a per newsgroup basis. Our interpretation to this is that the probability that an article might be accessed can be estimated better by looking at all the articles in the group than by just looking at one article.

Additionally, NEWSCACHE makes the life of Usenet administrators easier by providing the following new features without forcing the administrator to install a news server with a full newsfeed: provision of local newsgroups, transparent merging of multiple news servers into one virtual news server, and providing a virtual full feed over slow links where a full feed would not be possible.

These factors should make NEWSCACHE popular in the future. An increasing number of people already use NEWSCACHE including Internet service providers and NEWSCACHE is included in the Debian Linux distribution.

Acknowledgements

Thanks to Mehdi Jazayeri for encouraging us to continue this work and Michael Gschwind for his help in the preparation of this article and his comments on earlier drafts.

Availability

NEWSCACHE is available under the terms of the GNU Public License (GPL) from `http://www.infosys.tuwien.ac.at/NewsCache/`. Additionally, you can also test the current NEWSCACHE release by pointing your newsreader to the news server `newscache.infosys.tuwien.ac.at` on the standard NNTP port (119). NEWSCACHE is also distributed as a part of the Debian Linux distribution.

References

[1] Chip Salzenberg, Gene Spafford, and Mark Moraes. What is Usenet? ftp://rtfm.mit.edu/pub/usenet-by-group/news.admin.misc/What_is_Usenet%3F, November 1998.

[2] Brian Kantor and Phil Lapsley. Network News Transfer Protocol - A proposed standard for the stream-based transmission of news. RFC977, February 1986.

[3] Stan Barber. Network News Transport Protocol. Internet draft, December 1998. draft-ietf-nntpext-base-07.txt.

[4] Mark Horton and R. Adams. Standard for Interchange of USENET Messages. RFC1036, December 1987.

[5] Karl L. Swartz. Forecasting disk resource requirements for a Usenet server. In *Proceedings of the Seventh System Administration Conference (LISA '93)*, pages 195–202. USENIX, November 1993.

[6] Nick Christenson, David Beckemeyer, and Trent Baker. A scalable news architecture on a single spool. *;login:*, 22(3):41–45, June 1997.

[7] Rich Salz. InternetNews: Usenet Transport for Internet Sites. In *Proceedings of the Summer 1992 USENIX Conference*. USENIX, June 1992.

[8] Martin G. Rathmayer. Realisierung eines Bestellsystems für Newsgruppen an der TU Wien. *Pipeline*, (23), October 1997.

[9] James Fidell, Dale Ghent, Nathan J. Mehl, Chris van den Berg, and Stephen Zedalis. Frequently Asked Questions about the INN (InterNetNews) NNTP Server. http://www.blank.org/innfaq/.

[10] Thomas Gschwind. A Cache Server for News. Master's thesis, Technische Universität Wien, April 1997. http://www.infosys.tuwien.ac.at/NewsCache/.

[11] Heiko W. Rupp. A Protocol for the Transmission of Net News Articles over IP multicast, March 1998. Internet Draft, draft-rfced-exp-rupp-04.txt.

[12] Stan Barber. NNTP Reference Implementation. ftp://ftp.academ.com/pub/nntp1.5/nntp.1.5.12.2.tar.Z, January 1996.

[13] Geoff Collyer and Henry Spencer. News Need Not Be Slow. In *Proceedings of the Winter 1987 USENIX Technical Conference*, 1987.

[14] Internet Software Consortium. The InterNetNews NNTP Server. ftp://ftp.isc.org/isc/inn/inn-2.0.tar.gz, June 1998.

[15] Julian Assange and Luke Bowker. `NNTPCache`. http://www.nntpcache.org/.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, October 1994.

[17] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, July 1997.

Reducing the Disk I/O of Web Proxy Server Caches

Carlos Maltzahn and Kathy J. Richardson
Compaq Computer Corporation
Network Systems Laboratory
Palo Alto, CA
carlosm@cs.colorado.edu, kjr@pa.dec.com

Dirk Grunwald
University of Colorado
Department of Computer Science
Boulder, CO
grunwald@cs.colorado.edu

Abstract

The dramatic increase of HTTP traffic on the Internet has resulted in wide-spread use of large caching proxy servers as critical Internet infrastructure components. With continued growth the demand for larger caches and higher performance proxies grows as well. The common bottleneck of large caching proxy servers is disk I/O. In this paper we evaluate ways to reduce the amount of required disk I/O. First we compare the file system interactions of two existing web proxy servers, CERN and SQUID. Then we show how design adjustments to the current SQUID cache architecture can dramatically reduce disk I/O. Our findings suggest two that strategies can significantly reduce disk I/O: (1) preserve locality of the HTTP reference stream while translating these references into cache references, and (2) use virtual memory instead of the file system for objects smaller than the system page size. The evaluated techniques reduced disk I/O by 50% to 70%.

1 Introduction

The dramatic increase of HTTP traffic on the Internet in the last years has lead to the wide use of large, enterprise-level World-Wide Web proxy servers. The three main purposes of these web proxy servers are to control and filter traffic between a corporate network and the Internet, to reduce user-perceived latency when loading objects from the Internet, and to reduce bandwidth between the corporate network and the Internet. The latter two are commonly accomplished by caching objects on local disks.

Apart from network latencies the bottleneck of Web cache performance is disk I/O [2, 27]. An easy but expensive solution would be to just keep the entire cache in primary memory. However, various studies have shown that the Web cache hit rate grows in a logarithmic-like fashion with the amount of traffic and the size of the client population [16, 11, 8] as well as logarithmic-proportional to the cache size [3, 15, 8, 31, 16, 25, 9, 11] (see [6] for a summary and possible explanation). In practice this results in cache sizes in the order of ten to hundred gigabytes or more [29]. To install a server with this much primary memory is in many cases still not feasible.

Until main memory becomes cheap enough, Web caches will use disks, so there is a strong interest in reducing the overhead of disk I/O. Some commercial Web proxy servers come with hardware and a special operating system that is optimized for disk I/O. However, these solutions are expensive and in many cases not affordable. There is a wide interest in portable, low-cost solutions which require not more than standard off-the-shelf hardware and software. In this paper we are interested in exploring ways to reduce disk I/O by changing the way a Web proxy server application utilizes a general-purpose Unix file system using standard Unix system calls.

In this paper we compare the file system interactions of two existing web proxy servers, CERN [18] and SQUID [30]. We show how adjustments to the current

SQUID cache architecture can dramatically reduce disk I/O. Our findings suggest that two strategies can significantly reduce disk I/O: (1) preserve locality of the HTTP reference stream while translating these references into cache references, and (2) use virtual memory instead of the file system for objects smaller than the system page size. We support our claims using measurements from actual file systems exercised by a trace driven workload collected from proxy server log data at a major corporate Internet gateway.

In the next section we describe the cache architectures of two widely used web proxy servers and their interaction with the underlying file systems. We then propose a number of alternative cache architectures. While these cache architectures assume infinite caches we investigate finite cache management strategies in section 3 focusing on disk I/O. In section 4 we present the methodology we used to evaluate the cache structures and section 5 presents the results of our performance study. After discussing related work in section 6, we conclude with a summary and future work in section 7.

2 Cache Architectures of Web Proxy Servers

We define the *cache architecture* of a Web proxy server as the way a proxy server interacts with a file system. A cache architecture names, stores, and retrieves objects from a file system, and maintains application-level meta-data about cached objects. To better understand the impact of cache architectures on file systems we first review the basic design goals of file systems and then describe the Unix Fast File System (FFS), the standard file system available on most variants of the UNIX operating system.

2.1 File systems

Since the speed of disks lags far behind the speed of main memory the most important factor in I/O performance is *whether* disk I/O occurs at all ([17], page 542). File systems use memory caches to reduce disk I/O. The file system provides a *buffer cache* and a *name cache.* The buffer cache serves as a place to transfer and cache data to and from the disk. The name cache stores file and directory *name resolutions* which associate file and directory names with file system data structures that otherwise reside on disk.

The Fast File System (FFS) [20] divides disk space into *blocks* of uniform size (either 4K or 8K Bytes). These are the basic units of disk space allocation. These blocks may be sub-divided into *fragments* of 1K Bytes for small files or files that require a non-integral number of blocks. Blocks are grouped into *cylinder groups* which are sets of typically sixteen adjacent cylinders. These cylinder groups are used to map file reference locality to physically adjacent disk space. FFS tries to store each directory and its content within one cylinder group and each file into a set of adjacent blocks. The FFS does not guarantee such file layout but uses a simple set of heuristics to achieve it. As the file system fills up, the FFS will increasingly often fail to maintain such a layout and the file system gets increasingly *fragmented.* A fragmented file system stores a large part of its files in non-adjacent blocks. Reading and writing data from and to non-adjacent blocks causes longer seek times and can severely reduce file system throughput.

Each file is described by meta-data in the form of *inodes.* An inode is a fixed length structure that contains information about the size and location of the file as well as up to fifteen pointers to the blocks which store the data of the file. The first 12 pointers are direct pointers while the last three pointers refer to *indirect blocks,* which contain pointers to additional file blocks or to additional indirect blocks. The vast majority of files are shorter than 96K Bytes, so in most cases an inode can directly point to all blocks of a file, and storing them within the same cylinder group further exploits this reference locality.

The design of the FFS reflects assumption about file system workloads. These assumptions are based on studies of workloads generated by workstations [23, 24]. These workstation workloads and Web cache request workloads share many, but not all of the same characteristics. Because most of their behavior is similar, the file system works reasonably well for caching Web pages. However there are differences; and tweaks to the way cache objects map onto the file system produce significant performance improvements.

We will show in the following sections that some file system aspects of the workload characteristics generated by certain cache architectures can differ from usual workstation workloads. These different workload characteristics lead to poor file system performance. We will also show that adjustments to cache architectures can dramatically improve file system performance.

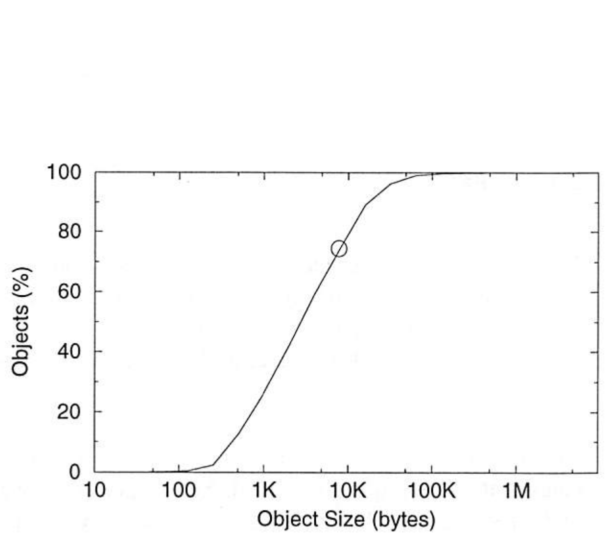


Figure 1: The dynamic size distribution of cached objects. The graph shows a cumulative distribution weighted by the number of objects. For example 74% of all object referenced have a size of equal or less than 8K Bytes.

2.2 File System Aspects of Web Proxy Server Cache Workloads

The basic function of a Web proxy server is to receive a request from a client, check whether the request is authorized, and serve the requested object either from a local disk or from the Internet. Generally, objects served from the Internet are also stored on a local disk so that future requests to the same object can be served locally. This functionality combined with Web traffic characteristics implies the following aspects of Web proxy server cache generated file system loads:

Entire Files Only Web objects are always written or read in their entirety. Web objects do change, but this causes the whole object to be rewritten; there are no incremental updates of cached Web objects. This is not significantly different than standard file system workloads where more than 65% of file accesses either read or write the whole file. Over 90% either read or write sequentially a portion of a file or the whole file [5]. Since there are no incremental additions to cached objects, it is likely that disk becomes more fragmented since there are fewer incremental bits to utilize small contiguous block segments.

Size Due to the characteristics of Web traffic, 74% of referenced Web objects are smaller than 8K Bytes. Figure 1 illustrates this by showing the distribution of the sizes of cached objects based on our HTTP traces, which are described later. This distribution is very similar to file characteristics. 8K Byte is a common system page size. Modern hardware supports the efficient transfer of system page sizes between disk and memory. A number of Unix File Systems use a file system block size of 8K Bytes and a fragment size of 1K Bytes. Typically the performance of the file system's fragment allocation mechanism has a greater impact on overall performance than the block allocation mechanism. In addition, fragment allocation is often more expensive than block allocation because fragment allocation usually involves a best-fit search.

Popularity The popularity of Web objects follows a *Zipf-like* distribution Ω/i^{α} (where $\Omega = (\sum_{i=1}^{N} 1/i^{\alpha})^{-1}$ and i is the ith most popular Web object) [15, 6]. The α values range from 0.64 to 0.83. Traces with homogeneous communities have a larger α value than traces with more diverse communities. The traces generally do not follow Zipf's law which states that $\alpha = 1$ [33]. The relative popularity of objects changes slowly (on the order of days and weeks). This implies that for any given trace of Web traffic, the first references to popular objects within a trace tend to occur early in the trace. The slow migration to new popular items allows for relatively static working set capture algorithms (see for example [28]). It also means that there is little or no working set behavior attributable to the majority of the referenced objects. File system references exhibit much more temporal locality; allocation and replacement policies need to react rapidly to working set changes.

Data Locality A large number of Web objects include links to embedded objects that are referenced in short succession. These references commonly refer to the same server and tend to have the same URL prefix. This is similar to the locality observed in workstation workloads which show that files accessed in short succession tend to be in the same file directory.

Meta-data Locality The fact that objects with similar names tend to be accessed in short succession means that information about those objects will also be referenced in short succession. If the information required to validate and access files is combined in the same manner as the file accesses it will exhibit temporal locality (many re-references within a short time period). The hierarchal directory structure of files systems tends to group related files together. The meta-data about those files and their access methods are stored in directory and inodes which end up being highly reused when accessing a group of files. Care is required to properly map Web objects to preserve the locality of meta-data.

Read/Write Ratio The hit rate of Web caches is low (30%-50%, see [15, 1, 31, 4]). Every cache hit involves a read of the cache meta-data and a read of the cached data. Every miss involves a read of the cache meta-data, a write of meta-data, and a write of the Web object. Since there are typically more misses than hits, the majority of disk accesses are writes. File systems typically have many more reads than writes [23]; writes require additional work because the file system data must be properly flushed from memory to disk. The high fraction of writes also causes the disk to quickly fragment. Data is written, removed and rewritten quickly; this makes it difficult to keep contiguous disk blocks available for fast or large data writes.

Redundancy Cached Web objects are (and should be) redundant; individual data items are not critical for the operation of Web proxy caches. If the cached data is lost, it can always be served from the Internet. This is not the case with file system data. Data lost before it is securely written to disk is irrecoverable. With highly reliable Web proxy servers (both software and hardware) it is acceptable to never actually store Web objects to disk, or to periodically store all Web objects to disk in the event of a server crash. This can significantly reduce the memory system page replacement cost for Web objects. A different assessment has to be made for the meta-data which some web proxy server use for cache management. In the event of meta-data loss, either the entire content of the cache is lost or has to be somehow rebuilt based on data saved on disk. High accessibility requirements might neither allow the loss of the entire cache nor time consuming cache rebuilds. In that case meta-data has to be handled similarly to file system data. The volume of meta-data is however much smaller than the volume of cached data.

2.3 Cache Architectures of Existing Web Proxy Servers

The following describes the cache architectures we are investigating in this paper. The first two describe the architectures of two widely used web proxy servers, CERN and SQUID. We then describe how the SQUID architecture could be changed to improve performance. All architectures assume infinite cache sizes. We discuss the management of finite caches in section 3.

2.3.1 CERN

The original web server `httpd` was developed at CERN and served as early reference implementation for World-Wide Web service. `httpd` can also be used as a web proxy server [18]. We refer to this function of httpd as "CERN".

CERN forks a new process for each request and terminates it after the request is served. The forked processes of CERN use the file system not only to store cached copies of Web objects but also to share meta-information about the content of the cache and to coordinate access to the cache. To find out whether a request can be served from the cache, CERN first translates the URL of the request into a URL directory and checks whether a *lock file* for the requested URL exists. The path of the URL directory is the result of mapping URL components to directories such that the length of the file path depends on the number of URL components. The check for a lock file requires the translation of each path component of the URL directory into an inode. Each translation can cause a miss in the file system's name cache in which case the translation requires information from the disk.

The existence of a lock file indicates that another CERN process is currently inserting the requested object into the cache. Locked objects are not served from the cache but fetched from the Internet without updating the cache. If no lock file exists, CERN tries to open a meta-data file in the URL directory. A failure to do so indicates a cache miss in which case CERN fetches the object from the Internet and inserts it into the cache thereby creating the necessary directories, temporary lock files, and meta-data file updates. All these operations require additional disk I/O in the case of misses in the file system's name and buffer cache. If the meta-data file exists and it lists the object file name as not expired, CERN serves the request from the cache.

2.3.2 SQUID

The SQUID proxy [30] uses a single process to eliminate CERN's overhead of process creation and termination. The process keeps meta-data about the cache contents in main memory. Each entry of the the meta-data maps a URL to a *unique file number* and contains data about the "freshness" of the cached object. If the meta-data does not contain an entry for the requested URL or the entry indicates that the cached copy is stale, the object is fetched from the Internet and inserted into the cache. Thus, with in-memory meta-data the disk is never touched to find out whether a request is a Web cache miss or a Web cache hit.

A unique file number n maps to a two-level file path that contains the cached object. The file path follows from the unique file number using the formula

$$(x, y, z) = (n \bmod l_1, n/l_1 \bmod l_2, n)$$

where (x, y, z) maps to the file path "x/y/z", and l_1 and l_2 are the numbers of first and second level directories. Unique file numbers for new objects are generated by either incrementing a global variable or reusing numbers from expired objects. This naming scheme ensures that the resulting directory tree is balanced. The number of first and second level directories are configurable to ensure that directories do not become too large. If directory objects exceed the size of a file block, directory look-up times increase.

2.4 Variations on the SQUID Cache Architecture

The main difference between CERN and SQUID is that CERN stores all state on disk while SQUID keeps a representation of the content of its cache (the meta-data) in main memory. It would seem straightforward to assume that CERN's architecture causes more disk I/O than SQUID's architecture. However, as we showed in [19], CERN's and SQUID's disk I/O are surprisingly similar for the same workload.

Our conjecture was that this is due to the fact that CERN's cache architecture preserves some of the locality of the HTTP reference stream, while SQUID's unique numbering scheme destroys locality. Although the CERN cache has a high file system overhead, the preservation of the spatial locality seen in the HTTP reference stream leads to a disk I/O performance comparable to the SQUID cache.

We have designed two alternative cache architectures for the SQUID cache that improve reference locality. We also investigated the benefits of circumventing the common file system abstractions for storing and retrieving objects by implementing *memory-mapped* caches. Memory mapped caches can reduce the number of file-system calls and effectively use large primary memories. However, memory-mapped caches also introduce more complexity for placement and replacement policies. We will examine several such allocation policies.

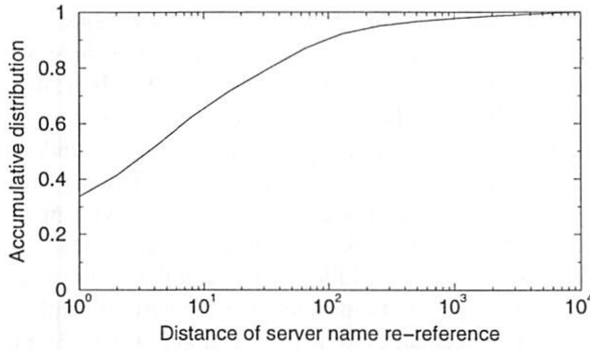


Figure 2: The locality of *server names* in an HTTP request stream. The data is based on an HTTP request stream with 495,662 requests (minus the first 100,000 to warm up the cache).

2.4.1 SQUIDL

We designed a modified SQUID cache architecture, SQUIDL, to determine whether a locality-preserving translation of an HTTP reference stream into a file system access stream reduces disk I/O. The only difference between SQUID and SQUIDL is that SQUIDL derives the URL directory name from the URL's host name instead of calculating a unique number. The modified formula for the file path of a cached object is now

$$(x, y, z) = (h(s) \wedge m_{l_1}, h(s) \wedge m_{l_2}, n)$$

where s is the host name of the requested URL, h is a hash function, $\wedge$ is the bitwise conjunction, m_{l_1} a bit mask for the first level directories, and m_{l_2} for the second level directories.

The rationale of this design is based on observation of the data shown in figure 2 (based on our HTTP traces, see below): the temporal locality of server names in HTTP references is high. One explanation for this is the fact that a large portion of HTTP requests are for objects that are embedded in the rendition of a requested HTML object. HTTP clients request these "in-lined" objects immediately after they parsed the HTML object. In most HTML objects all in-lined objects are from the same server. Since SQUIDL stores cached objects of the same server in the same directory, cache references to linked objects will tend to access the same directory. This leads to a burst of requests to the same directory and therefore increases the temporal locality of file system requests.

One drawback of SQUIDL is that a single directory may store many objects from a popular server. This can lead to directories with many entries which results in directory objects spanning multiple data blocks. Directory lookups in directory objects that are larger than one block can take significantly longer than directory lookups in single block directory objects [21]. If the disk cache is distributed across multiple file systems, directories of popular servers can put some file systems under a significantly higher workload than others. The SQUIDL architecture does produce a few directories with many files; for our workload only about 30 directories contained more than 1000 files. Although this skewed access pattern was not a problem for our system configuration, recent changes to SQUID version 2.0 [10, 13] implements a strategy that may be useful for large configurations. The changes balance file system load and size by allocating at most k files to a given directory. Once a directory reaches this user-configured number of files, SQUID switches to a different directory. The indexing function for this strategy can be expressed by

$$(x, y, z) = (n/(k * l_2), n/k \bmod l_2, n \bmod k)$$

where k is specified by the cache administrator. Notice that this formula poses an upper limit of $\mathit{max\_objs} = l_1 * l_2 * k$ objects that can be stored in the cache. Extensions to this formula could produce relatively balanced locality-preserving directory structures.

2.4.2 SQUIDM

One approach to reduce disk I/O is to circumvent the file system abstractions and store objects into a large memory-mapped file [22]. Disk space of the memory-mapped file is allocated once and access to the file are entirely managed by the virtual memory system. This has the following advantages:

Naming Stored objects are identified by the offset into the memory-mapped file which directly translates into a virtual memory address. This by-passes the overhead of translating file names into inodes and maintaining and storing those inodes.

Allocation The memory-mapped file is allocated once. If the file is created on a new file system, the allocated disk space is minimally fragmented which allows high utilization of disk bandwidth. As long as the file does not change in size, the allocated disk space will remain unfragmented. This one-time allocation also by-passes file system block and fragment allocation overhead [1]. Notice that memory-mapped files does not prevent *internal fragmentation*, *i.e.* the possible fragmentation of the content of the memory-mapped file due to application-level data management of the data stored in memory-mapped files. Since we assume infinite caches, internal fragmentation is not an issue here. See section 3 for the management of finite memory-mapped caches.

Paging Disk I/O is managed by virtual memory which takes advantage of hardware optimized for paging. The smallest unit of disk I/O is a system page instead of the size of the smallest stored object.

[1] This assumes that the underlying file system is not a log structured file system. File systems that log updates to data need to continually allocate new blocks and obliterate old blocks, thereby introducing fragmentation over time.

Thus, we expect that memory-mapping will benefit us primarily in the access of small objects by eliminating the opening and closing of small files. Most operating systems have limits on the size of memory-mapped files, and care must be taken to appropriately choose the objects to store in the limited space available. In the cache architecture SQUIDM we therefore chose the system page size (8K Byte) as upper limit. Over 70% of all object references are less or equal than 8K bytes (see figure 1 which is based on our HTTP traces). Objects larger than 8K Bytes are cached the same way as in SQUID.

To retrieve an object from a memory-mapped file we need to have its offset into the memory-mapped file and its size. In SQUIDM offset and size of each object are stored in in-memory meta-data. Instead of keeping track of the actual size of an object we defined five *segment sizes* (512, 1024, 2048, 4,096, or 8,192 Bytes). This reduces the size information from thirteen bits down to three bits. Each object is padded to the smallest segment size. In section 3 we will show more advantages of managing segments instead of object sizes.

These padded objects are contiguously written into the mapped virtual memory area in the order in which they are first referenced (and thus missed). Our conjecture was that this strategy would translate the temporal locality of the HTTP reference stream into spatial locality of virtual memory references.

We will show that this strategy also tends to concentrate very popular objects in the first few pages of the memory-mapped file; truly popular objects will be referenced frequently enough to be at the beginning of any reference stream. Clustering popular objects significantly reduces the number of page faults since those pages tend to stay in main memory. Over time, the set of popular references may change, increasing the page fault rate.

Algorithm 1 Algorithm to pack objects without crossing system page boundaries. The algorithm accepts a list of objects sizes of $\leq$ 8192 Bytes and outputs a list of offsets for packing each object without crossing system page boundaries (the size of a system page is 8192 Bytes).

```
proc packer(list_object_sizes) ≡
  One offset pointer for each segment size:
  512, 1024, 2048, 4096, 8192
  freelist := [0, 0, 0, 0, 0];
  offset_list := [];
  for i := 0 to length(list_of_object_sizes) − 1 do
     size := list_of_object_sizes[i];
     Determine segment size that fits object
     for segment := 0 to 4 do
        if size ≤ $2^{9+segment}$ then exit fi od;
     Find smallest available segment that fits
     for free_seg := segment to 4 do
        if freelist[free_seg] > 0 ∨ free_seg = 4
          then offset := freelist[free_seg];
               if free_seg = 4
                 Set 8192-pointer to next system page
                 otherwise mark free segment as taken
                 then freelist[4] := offset + 8192
                  else freelist[free_seg] := 0 fi;
               Update freelist with what is left
               for rest_seg := segment to free_seg − 1 do
                    new_offset := offset + $2^{9+rest\_seg}$;
                    freelist[rest_seg] := new_offset od;
               exit fi od;
     append(offset, offset_list)
  od;
  offset_list.
```

2.4.3 SQUIDML

The SQUIDML architecture uses a combination of SQUIDM for objects smaller than 8K Byte and SQUIDL for all other objects.

2.4.4 SQUIDMLA

The SQUIDMLA architecture combines the SQUIDML architecture with an algorithm to *align* objects in the memory mapped file such that no object crosses a page boundary. An important requirement of such an algorithm is that it preserves reference locality. We use a packing algorithm, shown in Algorithm 1 that for the given traces only slightly modifies the order in which objects are stored in the memory-mapped file. The algorithm insures that no object crosses page boundaries.

3 Management of Memory-mapped Web Caches

In the previous section we reasoned that storing small objects in a memory-mapped file can significantly reduce disk I/O. We assumed infinite cache size and therefore did not address replacement strategies. In this section we explore the effect of replacement strategies on disk I/O of finite cache architectures which use memory-mapped files.

Cache architectures which use the file system to cache objects to either individual files or one memory-mapped file are really two-level cache architectures: the first-level cache is the buffer cache in the primary memory and the second-level cache is the disk. However, standard operating systems generally do not support sufficient user-level control on buffer cache management to control primary memory replacement. This leaves us with the problem of replacing objects in secondary memory in such a way that disk I/O is minimized.

In the following sections we first review relevant aspects of system-level management of memory-mapped files. We then introduce three replacement algorithms and evaluate their performance.

3.1 Memory-mapped Files

A memory-mapped file is represented in the virtual memory system as a virtual memory object associated with a *pager*. A pager is responsible for filling and cleaning pages from and to a file. In older Unix systems the pager would operate on top of the file system. Because the virtual memory system and the file system used to be two independent systems, this led to the duplication of each page of a memory-mapped file. One copy would be stored in a buffer managed by the buffer cache and another in a page frame managed by the virtual memory. This duplication is not only wasteful but also leads to cache inconsistencies. Newer Unix implementations have a "unified buffer cache" where loaded virtual memory pages and buffer cache buffers can refer to the same physical memory location.

If access to a virtual memory address causes a page fault, the page fault handler is selecting a *target page* and passes control to the pager which is responsible for filling the page with the appropriate data. A *pager* translates the virtual memory address which caused the page fault into the memory-mapped file offset and retrieves the corresponding data from disk.

In the context of memory-mapped files, a page is *dirty* if it contains information that differs from the corresponding part of the file stored on disk. A page is *clean* if its information matches the information on the associated part of the file on disk. We call the process of writing dirty pages to disk *cleaning*. If the target page of a page fault is dirty it needs to be cleaned before it can be handed to the pager. Dirty pages are also cleaned periodically, typically every 30 seconds.

The latency of a disk transaction does not depend on the amount of data transferred but on disk arm repositioning and rotational delays. The file system as well as disk drivers and disk hardware are designed to minimize disk arm repositioning and rotational delays for a given access stream by reordering access requests depending on the current position of the disk arm and the current disk sector. However, reordering can only occur to a limited extent. Disk arm repositioning and rotational delays are still mainly dependent on the access pattern of the access stream and the disk layout.

Studies on file systems (*e.g.* [23]) have shown that the majority of file system access is a sequential access of logically adjacent data blocks. File systems therefore establish disk layout which is optimized for sequential access by placing logically adjacent blocks into physically adjacent sectors of the same cylinder whenever possible [21]. Thus, a sequential access stream minimizes disk arm repositioning and rotational delays and therefore reduces the latency of disk transactions.

If the entire memory-mapped file fits into primary memory, the only disk I/O is caused by periodic page cleaning and depends on the number of dirty pages per periodic page cleaning and the length of the period between page cleaning. The smaller the fraction of the memory-mapped file which fits into primary memory, the higher the number of page faults. Each page fault will cause extra disk I/O. If page faults occur randomly throughout the file, each page fault will require a separate disk I/O transaction. The larger the number of dirty pages the higher the likelihood that the page fault handler will choose dirty target pages which need to be cleaned before being replaced. Cleaning target pages will further increase disk I/O.

The challenge of using memory-mapped files as caches is to find replacement strategies that keep the number of page faults as low as possible, and that create an access stream as sequential as possible.

3.2 Cache Management

We are looking for cache management strategies which optimize hit rate but minimize disk I/O. We first introduce a strategy that requires knowledge of the entire trace. Even though this strategy is not practical it serves as an illustration on how to ideally avoid disk I/O. We then investigate the use of the most common replacement strategy, LRU and discuss its possible drawbacks. This motivates the design of a third replacement strategy which uses a combination of cyclic and frequency-based replacement.

3.3 Replacement strategies

Before we look at specific replacement algorithms it is useful to review an object replacement in terms of disk I/O. All replacement strategies are extensions of the SQUIDMLA cache architecture except the "future looking" strategy which is an extension of SQUIDML. The replacement strategies act on segments. Thus the size of an object is either 512, 1K, 2K, 4K, or 8K Bytes. For simplicity an object can replace another object of the same segment size only. We call objects to be replaced the *target object* and the page on which the target object resides, the *target object's page*. Notice that this is not the same as the *target page* which is the page to be replaced in a page fault. What disk I/O is caused by a object replacement depends on the following factors:

Whether the target object's page is loaded If the target object's page is already loaded in primary memory, no immediate disk I/O is necessary. Like in all other cases the replacement dirties the target object's page. All following factors assume that the target object's page is not loaded.

Object's size Objects of size 8K Bytes replace the entire content of the object's target page. If the object is of a smaller segment size the target object's page needs to be faulted into memory to correctly initialize primary memory.

Whether the target page is dirty If the target object's page needs to be loaded, it is written to a memory location of a target page (we assume the steady-state where loading a page requires to clear out a target page). If the target page is dirty it needs to be written to disk before it can be replaced.

The best case for a replacement is when the target object's page is already loaded. In the worst case a replacement case causes two disk I/O transactions: one to write a dirty page to disk, and another to fault in the target object's page for an object of a segment size smaller than 8K Bytes.

Beside the synchronous disk I/O there is also disk I/O caused by the periodic page cleaning of the operating system. If a replacement strategy creates a large number of dirty pages, the disk I/O of page cleaning is significant and can delay read and write system calls.

3.4 "Future-looking" Replacement

Our "future looking" strategy modifies the SQUIDML architecture to use a pre-computed placement table that is derived from *the entire trace*, including all future references. The intent is to build a "near optimal" allocation policy, while avoiding the computational complexity of implementing a perfect bin-packing algorithm, which would take non-polynomial time. The placement table is used to determine whether a reference is a miss or a hit, whether an object should be cached, and where it should be placed in the cache. We use the following heuristics to build the placement table:

1. All objects that occur in the workload are sorted by their popularity and all objects that are only referenced once are discarded, since these would never be re-referenced in the cache.
2. The remaining objects are sorted by descending order of their popularity. The packer algorithm of SQUIDMLA (see algorithm 1) is then used to generate offsets until objects cannot be packed without exceeding the cache size.
3. Objects which do not fit into the cache during the second step are then placed such that they replace the most popular object, and the time period between the first and last reference of the new object does not overlap with the time period between the first and last reference of the replaced object.

The goal of the third step is to place objects in pages that are likely to be memory resident but without causing extra misses. Objects that cannot be placed into the cache without generating extra misses to cached objects are dropped on the assumption that their low popularity will not justify extra misses to more popular objects.

3.5 LRU Replacement

The LRU strategy combines SQUIDMLA with LRU replacement for objects stored in the memory-mapped file. The advantage of this strategy is that it keeps popular objects in the cache. The disadvantage of LRU in the context of memory-mapped files is that it has no concept of collocating popular objects on one page and therefore tends to choose target objects on pages that are very likely not loaded. This has two effects: First it causes a lot of page faults since a large percentage of target objects are of smaller segment size than 8K. Second, the large number of page faults creates a large number of dirty pages which causes significant page cleaning overhead and also increases the likelihood of the worst case where a replacement causes two disk I/O transactions. A third disadvantage of LRU replacement is that the selection of a target page is likely to generate a mostly random access stream instead of a more sequential access stream.

3.6 Frequency-based Cyclic (FBC) Replacement

We now introduce a new strategy we call Frequency-based Cyclic (FBC) replacement. FBC maintains access frequency counts of each cached object and a target pointer that points to the first object that it considers for replacement. Which object actually gets replaced depends on the reference frequency of that object. If the reference frequency is equal or greater than C_{max}, the target pointer is advanced to the next object of the same segment size. If the reference frequency is less than C_{max}, the object becomes the target object for replacement. After replacing the object the target pointer is advanced to the next object. If the target pointer reaches the end of the cache it is reset to the beginning. Frequency counts are aged whenever the average reference count of all objects becomes greater than A_{max}. If the average value reaches this value, each frequency count c is reduced to $\lceil c/2 \rceil$. Thus, in the steady state the sum of all reference counts stay between $N \times A_{max}/2$ and $N \times A_{max}$ (where N is the number of cached objects). The ceiling function is necessary because we maintain a minimum reference count of one. This aging mechanism follows the approach mentioned in [26, 12].

Since Web caching has a low hit rate, most cached objects are never referenced again. This in turns means that most of the time, the first object to which the target pointer points becomes the target object. The result is an almost sequential creation of dirty pages and page faults which is likely to produce a sequential access stream. Skipping popular pages has two effects. Firstly, it avoids replacing popular objects, and secondly the combination of cyclic replacement and aging factors out references to objects that are only popular for a short time. Short-term popularity is likely to age away within a few replacement cycles.

The two parameters of FBC, C_{max} and A_{max} have the following intuitive meaning. C_{max} determines the threshold below which a page is replaced if cyclic replacement points to it (otherwise it is skipped). For high C_{max} the hit rate suffers because more popular objects are being replaced. For low C_{max} more objects are skipped and the access stream becomes less sequential. With the Zipf-like distribution of object popularity, most objects are only accessed once. This allows low values for C_{max} without disturbing sequential access. A_{max} determines how often objects are aged. For high A_{max} aging takes place at a low frequency which leaves short-term-popular objects with high reference counts for a longer period of time. Low A_{max} values culls out short-term popularity more quickly but also make popular objects with a low but stable reference frequency look indistinguishable from less popular objects. Because of the Zipf-like distribution of object popularity, a high A_{max} will introduce only a relatively small set of objects that are popular for a short term only.

4 Experimental Methodology

In order to test these cache architectures we built a *disk workload generator* that simulates the part of a Web cache that accesses the file system or the virtual memory. With minor differences, the simulator performs the same disk I/O activity that would be requested by the proxy. However, by using a simulator, we simplified the task of implementing the different allocation and replacement policies and greatly simplified our experiments. Using a simulator rather than a proxy allows us to use traces of actual cache requests without having to mimic the full Internet. Thus, we could run repeatable measurements on the cache component we were studying – the disk I/O system.

The workload generators are driven by actual HTTP Web proxy server traces. Each trace entry consists of a URL and the size of the referenced object. During an experiment a workload generator sequentially processes each trace entry – the generator first determines whether a cached object exists and then either "misses" the object into the cache by writing data of the specified size to the appropriate location or "hits" the object by reading the corresponding data. Our workload generators process requests sequentially and thus our experiments do not account for the fact that the CERN and SQUID architecture allow multiple files to be open at the same time and that access to files can be interleaved. Unfortunately this hides possible file system locking issues.

We ran all infinite cache experiments on a dedicated Digital Alpha Station 250 4/266 with 512M Byte main memory. We used two 4G Byte disks and one 2G Byte disk to store cached objects. We used the UFS file system that comes with Digital Unix 4.0 for all experiments except those that calibrate the experiments in this paper to those in earlier work. The UFS file system uses a block size of 8192 Bytes and a fragment size of 1024 Bytes. For the comparison of SQUID and CERN we used Digital's Advanced File System (AdvFS) to validate our experiments with the results reported in [19].

UFS cannot span multiple disks so we needed a separate file system for each disk. All UFS experiments measured SQUID derived architectures with 16 first-level directories and 256 second-level directories. These 4096 directories were distributed over the three file systems, 820 directories on the 2G Byte disk and 1638 directories on each of the 4G Byte disks. When using memory-mapped caches, we placed 2048 directories on each 4G Byte disk and used the 2G Byte disk exclusively for the memory-mapped file. This also allowed us to measure memory-mapped-based caching separately from file-system-based caching.

We ran all finite cache experiments on a dedicated Digital Alpha Station 3000 with 64M Byte main memory and a 1.6G Byte disk. We set the size of the memory-mapped file to 160M Bytes. This size ensures ample exercise of the Web cache replacement strategies we are testing. The size is also roughly six times the size of the amount of primary memory used for memory-mapping (about 24M Bytes; the workload generator used 173M Bytes of virtual memory and the resident size stabilized at 37M Bytes). This creates sufficient pressure on primary memory to see the influence of the tested replacement strategies on buffer cache performance.

For the infinite cache experiments we used traces from Digital's corporate gateway in Palo Alto, CA, which runs two Web proxy servers that share the load by using round-robin DNS. We picked two consecutive weekdays of one proxy server and removed every non-HTTP request, every HTTP request with a reply code other than

200 ("OK"), and every HTTP request which contain "?" or "cgi-bin". The resulting trace data consists of 522,376 requests of the first weekday and 495,664 requests of the second weekday. Assuming an infinite cache, the trace leads to a hit rate of 59%. This is a high hit rate for a Web proxy trace; it is due to the omission of non-cacheable material and the fact that we ignore object staleness.

For the finite cache experiments we used the same traces. Because we are only interested in the performance of memory-mapped files, we removed from the traces all references to objects larger than 8K Bytes since these would be stored as individual files and not in the memory-mapped file. As parameters for FBC we used $C_{max} = 3$ and $A_{max} = 100$.

Each experiment consisted of two phases: the first *warmup* phase started with a newly initialized file system and newly formatted disks on which the workload generator ran the requests of the first day. The second *measurement* phase consisted of processing the requests of the following day using the main-memory and disk state that resulted from the first phase. All measurements are taken during the second phase using the trace data of the second weekday. Thus, we can directly compare the performance of each mimicked cache architecture by the absolute values of disk I/O.

We measured the disk I/O of the simulations using AdvFS with a tool called `advfsstat` using the command `advfsstat -i 1 -v 0 cache_domain`, which lists the number of reads and writes for every disk associated with the file domain. For the disk I/O of the simulations using UFS we used `iostat`. We used `iostat rz3 rz5 rz6 1`, which lists the bytes and transfers for the three disks once per second. Unfortunately, `iostat` does not segregate the number of reads and writes.

5 Results

We first compared the results of our cache simulator to our prior work to determine that the simulator exercised the disk subsystem with similar results to the actual proxy caches. We measured the disk I/O of the two workload generators that mimic CERN and SQUID to see whether the generator approach reproduces the same relative disk I/O as observed on the real counterparts [19]. As Figure 3 shows, the disk I/O is similar when using the CERN and SQUID workload generators. This agrees with our earlier measurements showing that CERN and SQUID make similar use of the disk subsystem. The

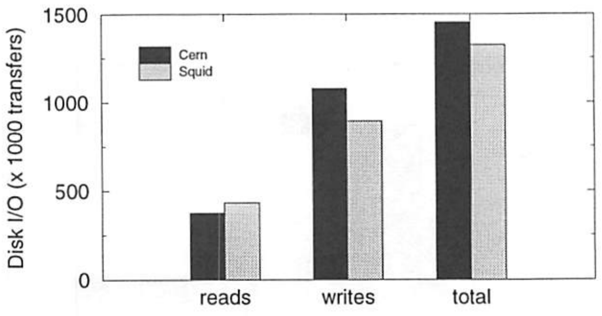


Figure 3: Disk I/O of the workload generators mimicking CERN and Squid. The measurements were taken from an AdvFS. In [19] we observed that The disk I/O of CERN and SQUID is surprisingly similar considering that SQUID maintains in-memory meta-data about its cache content and CERN does not. Our workload generators reproduce this phenomena.

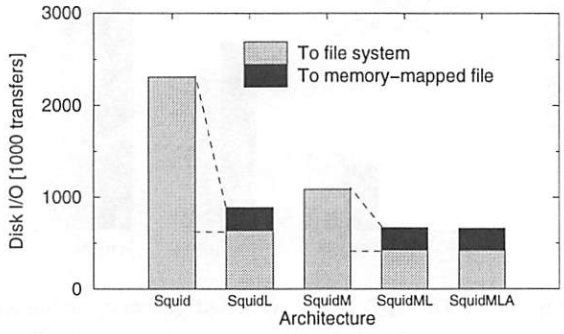
measurements were taken using the AdvFS file system because the Web proxy servers measured in [19] used that file system. The AdvFS utilities allowed us to distinguish between reads and writes. The data shows that only a third of all disk I/O are reads even though the cache hit rate is 59%.

Our traces referenced less than 8G Bytes of data, and thus we could conduct measurements for "infinite" caches with the experimental hardware. Figure 4 shows the number of disk I/O transactions and the duration of each trace execution for each of the architectures.

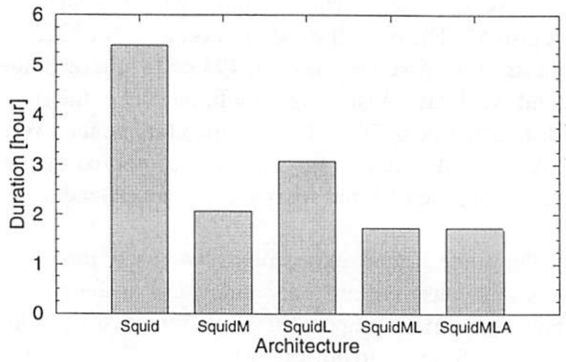
Comparing the performance of SQUID and SQUIDL shows that simply changing the function used to index the URL reduces the disk I/O by $\approx 50\%$.

By comparing SQUID and SQUIDM we can observe that memory mapping all small objects not only improves locality but produces a greater overall improvement in disk activity: SQUIDM produces 60% fewer disk I/O. Recall that SQUIDM stores all objects of size ≤ 8192 in a memory-mapped file and all larger objects in the same way as SQUID. As shown in Figure 1, about 70% of all references are to objects ≤ 8192. Thus, the remaining 30% of all references go to objects stored using the SQUID caching architecture. If we assume that these latter references account for roughly the same disk I/O in SQUIDM as in SQUID, none of the benefits come from these 30% of references. This means that there is an 85% savings generated off of the remaining 70% of SQUID's original disk I/O. Much of the savings occurs because writes to the cache are not immediately committed to the disk, allowing larger disk transfers.

An analogous observation can be made by comparing



(a)



(b)

Figure 4: Disk I/O of SQUID derived architectures. Graph (a) breaks down the disk I/O into file system traffic and memory-mapped file traffic. Graph (b) compares compares the duration of the measurement phase of each experiment

SQUIDML with SQUIDL. Here, using memory-mapping cache saves about 63% of SQUIDL's original disk I/O for objects of size $\leq$ 8192. The disk I/O savings of SQUIDM and SQUIDML are largely due to larger disk transfers that occur less frequently. The average I/O transfer size for SQUIDM and SQUIDML is 21K Bytes to the memory-mapped file, while the average transfer sizes to SQUID and SQUIDL style files are 8K Bytes and 10K Bytes, respectively.

The SQUIDMLA architecture strictly aligns segments to page boundaries such that no object spans two memory pages. This optimization would be important for purely disk-based caches, since it reduces the number of "read-modify-write" disk transactions and the number of transactions to different blocks. The results show that this alignment has no discernible impact on disk I/O. We found that SQUIDM and SQUIDML places 32% of the cached objects across page boundaries (30% of the cache hits were to objects that are crossing page boundaries).

Figure 5 confirms our conjecture that popular objects tend to be missed early. 70% of the references go to 25% of the pages to which the cache file is memory-mapped. Placing objects in the order of misses leads therefore to a higher page hit rate.

We evaluate the performance of each replacement strategy by the amount of disk I/O and the cache hit rate. As expected, the LRU replacement policy causes the highest number of disk transactions during the measurement phase. The future-looking policy shows that the actual working set at any point in time is small, and that accurate predictions of page reuse patterns would produce high hit rates on physical memory sized caches. Figure 6 show that the frequency-based cyclic replacement causes less disk I/O than LRU replacement without changing the hit rate. The figure also shows the time savings caused by reduced disk I/O. The time savings are greater than the disk I/O savings which indicates a more sequential access stream where more transactions access the same cylinder and therefore do not require disk arm repositioning.

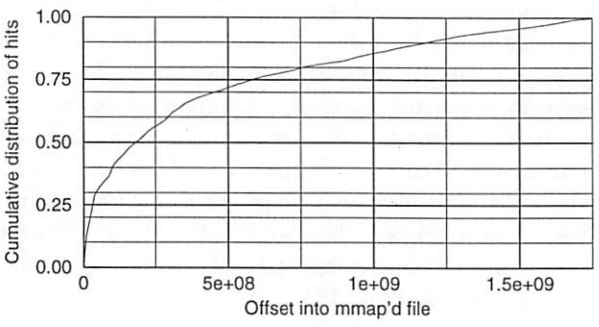
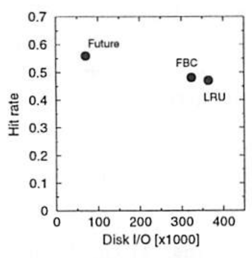


Figure 5: The cumulative hit distribution over the virtual address space of the memory-mapped cache file. 70% of the hits occur in the first quarter of the memory-mapped cache file.

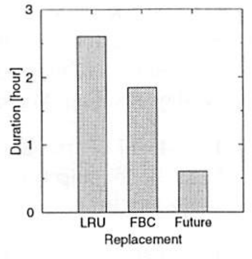
6 Related Work

There exist a large body of research work on application-level buffer control mechanisms. The external pager in Mach [32] allow users to implement paging between primary memory and disk. Cao *et al.* investigate a mechanism to allow users to manage page replacement without degrading overall performance in a multi-programmed system [7].

Glass and Cao propose and evaluate in [14] a kernel-



(a)



(b)

Figure 6: Disk I/O and hit rate tradeoffs of different replacement strategies. The graph (a) plots disk I/O against hit rate of the three replacement experiments. Note that *lower* x-values are better than higher ones. The graph (b) shows the duration of each experiment.

level page replacement strategy SEQ that detects long sequences of page faults and applies most-recently-used replacement to those sequences. The frequency-based cyclic web cache replacement strategy proposed above is specifically designed to generate more sequential page faults. We are currently investigating the combined performance of SEQ buffer caches and cyclic-frequency-based web caches.

7 Conclusions and Future Research

We showed that some design adjustments to the SQUID architecture can result in a significant reduction of disk I/O. Web workloads exhibit much of the same reference characteristics as file system workloads. As with any high performance application it is important to map file system access patterns so that they mimic traditional workloads to exploit existing operating caching features. Merely maintaining the first level directory reference hierarchy and locality when mapping web objects to the file system improved system the meta-data caching and reduced the number of disk I/O's by 50%.

The size and reuse patterns for web objects are also similar. The most popular pages are small. Caching small objects in memory mapped files allows most of the hits to be captured with no disk I/O at all. Using the combination of locality-preserving file paths and memory-mapped files our simulations resulted in disk I/O savings of over 70%.

Very large memory mapped caches significantly reduce the number of disk I/O requests and produce high cache hit rates. Future work will concentrate on replacement techniques that further reduce I/O from memory mapped caches while maintaining high hit rates. Our experience with the future-looking algorithm shows that there is an additional 10% reduction possible. Our experience with the LRU algorithm suggests that managing small memory mapped caches requires a tighter synchronization with the operating system memory management system. Possibilities include extensions that allow application management of pages, or knowledge of the current state of page allocation.

Our experiments do not account for overhead due to staleness of cached objects and cache replacement strategies. However, our results should be encouraging enough to motivate an implementation of some of the described cache architectures in an experimental version of SQUID.

References

[1] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Step hen Williaams, and Edward A. Fox. Caching proxies: Limitations and potentials. Available on the World-Wide Web at http://ei.cs.vt.edu/~succeed/WWW4/WWW4.html.

[2] Jussara Almeida and Pei Cao. Measuring Proxy Performance with the Wisconsin Proxy Benchmark. Technical Report 1373, Computer Science Department, University of Wisconsin-Madison, April 13 1998.

[3] Virgilio Almeida, Azer Bestavros, Mark Crovella, , and Adriana de Oliveira. Characterizing Reference Locality in the WWW. In *IEEE PDIS'96: The International Conference in Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1996. IEEE.

[4] Martin F. Arlitt and Carey L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *ACM Sigmetrics '96*, pages 126–137, Philadelphia, PA, May 23-26 1996. ACM Sigmetrics, ACM.

[5] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proceedings fo the Symposium on Operating Systems Principles (SOSP)*, pages 198–212. ACM, October 1991.

[6] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. Technical Report 1371, Computer Sciences Dept, Univ. of Wisconsin-Madison, April 1998.

[7] Pei Cao, Edward W. Felten, and Kai Li. Application-Controlled File Caching Policies. In *USENIX Summer 1994 Technical Conference*, 1994.

[8] Pei Cao and Sandy Irani. Cost-Aware Web Proxy Caching. In *Usenix Symposium on Internet Technologies and Systems (USITS)*, pages 193–206, Monterey, CA, USA, December 8-11 1997. Usenix.

[9] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW Client-based Traces. Technical Report BU-CS-95-010, Computer Science Department, Boston University, July 18 1995.

[10] Arjan de Vet. Squid new disk storage scheme. Available on the World Wide Web at http://www.iaehv.nl/users/devet/squid/new_store/, November 17 1997.

[11] Brad Duska, David Marwood, and Michael J. Feeley. The Measured Access Characteristics of World-Wide Web Client Proxy Caches. In *Usenix Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, USA, December 8-11 1997. Usenix.

[12] W. Effelsberg and T. Haerder. Principles of Database Buffer Management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.

[13] Stewart Forster. Personal Communication. Description of disk I/O in Squid 1.2, August 14 1998.

[14] Gideon Glass and Pei Cao. Adaptive Page Replacement Based on Memory Reference Behavior. In *ACM Sigmetrics 97*, pages 115–126, Seattle, WA, June 1997.

[15] Steven Glassman. A caching relay for the world-wide web. In *First International World-Wide Web Conference*, pages 69–76. W3O, May 1994.

[16] Steven D. Gribble and Eric A. Brewer. System design issues for Internet Middleware Services: Deductions from a large client trace. In *Usenix Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, USA, December 8-11 1997. Usenix.

[17] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (2nd edition)*. Morgan Kaufmann, San Francisco, CA, 1996.

[18] Ari Luotonen, Henrik Frystyk Nielsen, and Tim Berners-Lee. CERN httpd 3.0A. Available on the World-Wide Web at http://www.w3.org/pub/WWW/Daemon/, July 15 1996.

[19] Carlos Maltzahn, Kathy Richardson, and Dirk Grunwald. Performance Issues of Enterprise Level Proxies. In *SIGMETRICS '97*, pages 13–23, Seattle, WA, June 15-18 1997. ACM.

[20] Marshall K. McKusick, William N. Joy, Samual J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[21] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, Reading, Massachusetts, 1996.

[22] Jeff Mogul. Personal communication. Idea to mmap a large file for objects less or equal than system page size, August 1996.

[23] John Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. Technical Report UCB/CSD 85/230, University of California, Berkeley, April 1985.

[24] Kathy J. Richardson. I/O Characterization and Attribute Caches for Improved I/O System Performance. Technical Report CSL-TR-94-655, Stanford University, Dec 1994.

[25] Luigi Rizzo and Lorenzo Vicisano. Replacement Policies for a Proxy Cache. Technical Report RN/98/13, University College London, Department of Computer Science, 1998.

[26] John T. Robinson and Murthy V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *SIGMETRICS'90*, pages 134–142, Boulder, CO, May 22-25 1990. ACM.

[27] Alex Rousskov and Valery Soloviev. A Performance Study of the Squid Proxy on HTTP/1.0. *to appear in World-Wide Web Journal*, Special Edition on WWW Characterization and Performance Evaluation, 1999.

[28] Alex Rousskov, Valery Soloviev, and Igor Tatarinov. Static Caching. In *2nd International Web Caching Workshop (WCW'97)*, Boulder, Colorado, June 9-10 1997.

[29] Squid Users. Squid-users mailing list archive. Available on the World-Wide Web at http://squid.nlanr.net/Mail-Archive/squid-users/hypermail.html, August 1998.

[30] Duane Wessels. Squid Internet Object Cache. Available on the World-Wide Web at http://squid.nlanr.net/Squid/, May 1996.

[31] Stephen Williams, Marc Abrams, Charles R. Standridge, Ch aleb Abdulla, and Edward A. Fox. Removal policies in network caches for world-wide web documents. In *ACM SIGCOMM*, pages 293–305, Stanford, CA, August 1996.

[32] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *11th Symposium on Operating systems Principles*, pages 63–76, November 1987.

[33] George Kingsley Zipf. Relative Frequency as a Determinant of Phonetic Change. *reprinted from Harvard Studies in Classical Philiology*, XL, 1929.

An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme

Jongmoo Choi† Sam H. Noh‡ Sang Lyul Min† Yookun Cho†

†*Department of Computer Engineering*
Seoul National University
Seoul 151-742, Korea
http://ssrnet.snu.ac.kr/~{choijm,cho}
http://archi.snu.ac.kr/~symin

‡*Department of Computer Engineering*
Hong-Ik University
Seoul 121-791, Korea
http://www.cs.hongik.ac.kr/~noh

Abstract

In this paper, we propose a new adaptive buffer management scheme called DEAR (DEtection based Adaptive Replacement) that automatically detects the block reference patterns of applications and applies different replacement policies to different applications based on the detected reference pattern. The proposed DEAR scheme uses a periodic process. Detection is made by associating block attribute values such as backward distance and frequency gathered at the $(i-1)$-th invocation with forward distances of blocks referenced between the $(i-1)$-th and i-th invocations. We implemented the DEAR scheme in FreeBSD 2.2.5 and measured its performance using several real applications. The results show that compared with the LRU buffer management scheme, the proposed scheme reduces the number of disk I/Os by up to 51% (with an average of 23%) and the response time by up to 35% (with an average of 12%) in the case of single application executions. For multiple applications, the proposed scheme reduces the number of disk I/Os by up to 20% (with an average of 12%) and the overall response time by up to 18% (with an average of 8%).

1 Introduction

The speed gap between processors and disks continues to increase as VLSI technologies advance at an enormous rate. This speed gap has resulted in disk I/O becoming a serious performance bottleneck for many computer systems [1, 2]. Hence, the role of the buffer cache located in main memory and managed by the operating system is becoming increasingly important. Judicious use of the buffer cache can improve the response time of individual applications and also the throughput of the system by reducing the number of disk I/Os. To this end, study of effective block replacement policies has been the focus of much research both in the systems and database areas [3, 4, 5, 6, 7, 8].

There also have been a number of studies on predicting future access for prefetching purposes [9, 10, 11]. They make use of past access history to predict files or blocks that are likely to be referenced in the near future. This prediction information is used to issue prefetch requests in order to hide disk I/O latency.

Recently, buffer management schemes based on user-level hints such as application-controlled file caching [12] and informed prefetching and caching [13] have been proposed. User-level hints in these schemes provide information about which blocks are good candidates for replacement, allowing different replacement policies to be applied to different applications.

However, to obtain user-level hints, users need to accurately understand the characteristics of block reference patterns of applications. This requires considerable effort from users limiting its applicability. For a sequential reference pattern, a simple heuristic method can be used to detect the pattern and the MRU replacement policy can be used to improve the buffer cache performance [14]. Such hints can also be obtained by the compiler for implicit I/Os to manage paged virtual memory [15]. As we will see later, our proposed approach is complementary to this approach since our approach is targeted for explicit I/Os.

In this paper, we propose a new buffer management scheme that we call DEAR (DEtection based Adaptive Replacement). Without any user intervention, the DEAR scheme detects the reference pattern of each application and classifies the pattern as sequential, looping, temporally-clustered, or probabilistic. After the detection, the scheme applies an appropriate replacement policy to the application. As the reference pattern of an application may change during its execution, the DEAR scheme periodically detects the reference pattern and applies a different replacement policy, if necessary.

We implemented the DEAR scheme in FreeBSD 2.2.5 and evaluated its performance with several real applications. The scheme is implemented at the kernel level without any modification to the system call interface, so the applications may run as-is. Performance measurements with real applications show that in the case of single application executions the DEAR scheme reduces the number of disk I/Os by up to 51% (with an average of 23%) and the response time by up to 35% (with an average of 12%), compared with the LRU buffer management scheme in FreeBSD. For multiple applications, the reduction in the number of disk I/Os is by up to 20% (with an average of 12%) while the reduction in the overall response time is by up to 18% (with an average of 8%).

We also compared the performance of the DEAR scheme with that of application-controlled file caching [12] through trace-driven simulations with the same set of application traces used in [12]. The results showed that the DEAR scheme without any use-level hints performs comparably to application-controlled file caching for the traces considered.

The rest of the paper is organized as follows. In Section 2, we explain the DEAR scheme in detail. Then, we describe the implementation of the DEAR scheme in FreeBSD in Section 3. In Section 4, we evaluate the performance of the DEAR scheme. Finally, we conclude with a summary and a discussion of future work in Section 5.

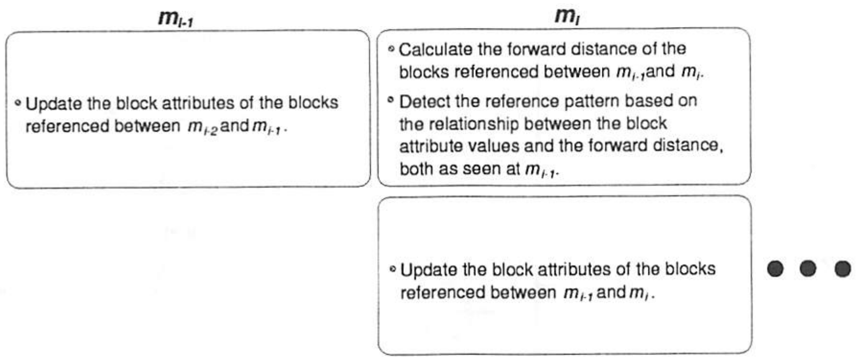
2 The DEAR Scheme

Recent research has shown that most applications show regular block reference patterns and that these patterns vary depending on the nature of the application. For example, a large class of scientific applications show a looping reference pattern where blocks are referenced repeatedly with regular intervals [16]. On the other hand, many database applications show a probabilistic reference pattern with different probabilities for index blocks and data blocks [17]. Unix applications tend to show either a sequential or a temporally-clustered reference pattern [12, 18]. Applications that deal with continuous media generally show a sequential or a looping reference pattern [19].

From these observations, we classify an application's reference pattern into one of the following: sequential, looping, temporally-clustered, or probabilistic reference pattern [20]. In the proposed DEAR scheme, the detection of an application's reference pattern is made by associating attributes of blocks with their forward distances, which are defined as the time intervals between the current time and the times of the next references. An attribute of a block can be anything that can be obtained from its past reference behavior including backward distance, frequency, inter-reference gap (IRG) [6], and k-th backward distance [4]. In this paper, we consider only two block attribute types: backward distance, which is the time interval between the current time and the time of the last reference[1], and frequency, which is the number of past references to the block.

The detection is performed by a monitoring process that is invoked periodically. At the time of its i-th invocation (we denote this time by m_i), the monitoring process calculates the forward distances (as seen from the standpoint of m_{i-1}) of the blocks referenced between m_{i-1} and m_i. From the block attribute values of those blocks, also as seen from the standpoint of m_{i-1}, the monitoring process builds two ordered lists using those blocks, one according to backward distance and the other according to frequency. Each ordered list is divided into a fixed number of sublists of equal size. Based on the relationship between the attribute value of each sublist and the average forward distance of blocks in the sublist, the block reference pattern of the application is deduced.

[1]In this paper, we assume that the (virtual) time is incremented at each block reference.



m_{i-1}
m_i
• Update the block attributes of the blocks referenced between m_{i-2} and m_{i-1}.
• Calculate the forward distance of the blocks referenced between m_{i-1} and m_i.
• Detect the reference pattern based on the relationship between the block attribute values and the forward distance, both as seen at m_{i-1}.
• Update the block attributes of the blocks referenced between m_{i-1} and m_i.

Figure 1: Detection process: two-stage pipeline with one-level look-behind.

After the detection, the block attributes of the blocks referenced between m_{i-1} and m_i are updated for the next detection at m_{i+1}. As shown in Figure 1, the detection process is essentially a two-stage pipeline with one-level look-behind since the detection at m_i is made based on the relationship between the block attribute values and the forward distance at m_{i-1}.

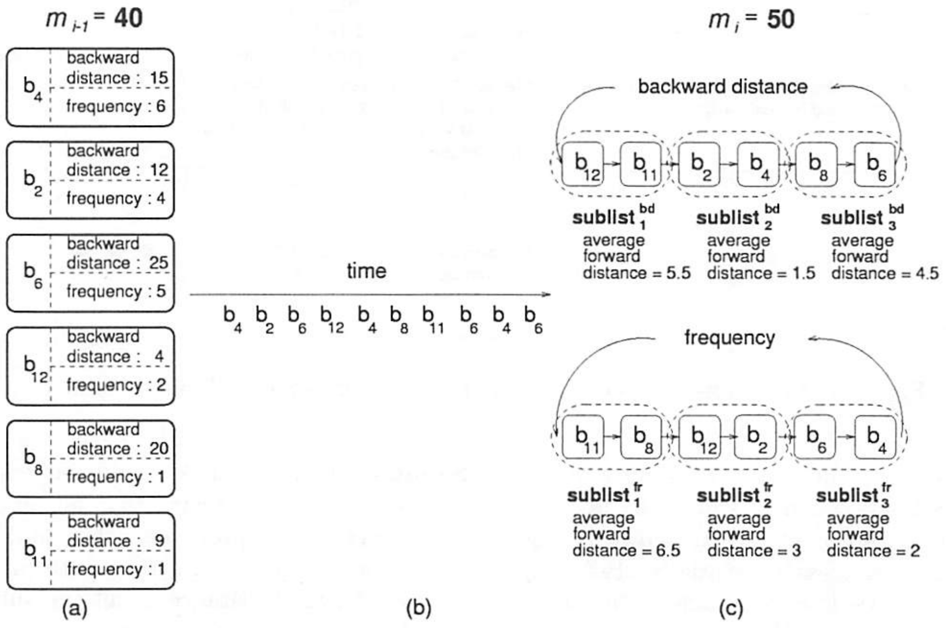
As an example, consider Figure 2. Assume that the period of the monitoring process (i.e., detection period) is 10 as measured in the number of block references made by the associated application. Also assume that between $m_{i-1} = 40$ and $m_i = 50$, blocks b_4, b_2, b_6, b_{12}, b_4, b_8, b_{11}, b_6, b_4, and b_6 were referenced in the given order (see Figure 2-(b)). Note that there are 10 block references since the detection period is 10. Finally, assume that at m_{i-1} the backward distance and frequency of the six distinct blocks b_4, b_2, b_6, b_{12}, b_8, b_{11} were 15, 12, 25, 4, 20, 9 and 6, 4, 5, 2, 1, 1, respectively (see Figure 2-(a)). Note that these distinct blocks have forward distances of 1, 2, 3, 4, 6, 7, respectively as seen at m_{i-1}. From the information about the block attribute values and the forward distance as seen at m_{i-1}, at m_i the DEAR scheme constructs two ordered lists, one according to backward distance and the other according to frequency (see Figure 2-(c)). Each list is divided into a number of sublists of equal size (3 sublists of size 2, in this example). Then various rules for detecting reference patterns, which are explained below, are applied to the two lists. In this particular example, blocks with higher frequency have smaller forward distance, which allows us to deduce that the block reference pattern of the given application follows a probabilistic reference pattern. The detection rules for all the reference patterns we consider are as follows:

Sequential Pattern: A sequential reference pattern has the property that all blocks are referenced one after the other and never referenced again. In this pattern, the average forward distance of all the sublists is ∞. Therefore, a reference pattern is sequential if $\mathbf{Avg\_fd}(sublist_1^{bd}) = \mathbf{Avg\_fd}(sublist_2^{bd}) = \cdots = \mathbf{Avg\_fd}(sublist_1^{fr}) = \mathbf{Avg\_fd}(sublist_2^{fr}) = \cdots = \infty$ where $sublist_i^{bd}$ and $sublist_i^{fr}$ are the i-th sublist for the backward distance and frequency block attribute types, respectively, and $\mathbf{Avg\_fd}(sublist)$ is the average forward distance of blocks in $sublist$.

Looping Pattern: A looping reference pattern has the property that blocks are referenced repeatedly with a regular interval. In this pattern, a block with a larger backward distance has a smaller forward distance. Therefore, a reference pattern is looping if the following relationship holds: if $i < j$ then $\mathbf{Avg\_fd}(sublist_i^{bd}) > \mathbf{Avg\_fd}(sublist_j^{bd})$.

Temporally-clustered Pattern: A temporally-clustered reference pattern has the property that a block referenced more recently will be referenced sooner in the future. Thus, a block with a smaller backward distance has a smaller forward distance. Therefore, a reference pattern is temporally-clustered if the following relationship holds: if $i < j$ then $\mathbf{Avg\_fd}(sublist_i^{bd}) < \mathbf{Avg\_fd}(sublist_j^{bd})$.

Probabilistic Pattern: A probabilistic reference pattern has a non-uniform block reference behavior that can be modeled by the Independent Reference Model (IRM) [21]. Each block b_i has a stationary probability p_i and all blocks are independently referenced with the associated probabilities. Under the stationary and independent condition, the expected forward



$m_{i-1} = 40$
$m_i = 50$
b_4 backward distance : 15 frequency : 6
b_2 backward distance : 12 frequency : 4
b_6 backward distance : 25 frequency : 5
b_{12} backward distance : 4 frequency : 2
b_8 backward distance : 20 frequency : 1
b_{11} backward distance : 9 frequency : 1
time
b_4 b_2 b_6 b_{12} b_4 b_8 b_{11} b_6 b_4 b_6
backward distance
b_{12} b_{11} b_2 b_4 b_8 b_6
$sublist_1^{bd}$ average forward distance = 5.5
$sublist_2^{bd}$ average forward distance = 1.5
$sublist_3^{bd}$ average forward distance = 4.5
frequency
b_{11} b_8 b_{12} b_2 b_6 b_4
$sublist_1^{fr}$ average forward distance = 6.5
$sublist_2^{fr}$ average forward distance = 3
$sublist_3^{fr}$ average forward distance = 2
(a)
(b)
(c)

Figure 2: Example of block reference pattern detection.

distance of b_i is proportional to $1/p_i$. Thus, a block with a higher frequency has a smaller forward distance. Therefore, a reference pattern is probabilistic if the following relationship holds: if $i < j$ then $\mathbf{Avg\_fd}(sublist_i^{fr}) > \mathbf{Avg\_fd}(sublist_j^{fr})$.

In the DEAR scheme, different replacement policies are used for different applications depending on the detected reference pattern. For the sequential and looping reference patterns, the MRU replacement policy is used where the block with the smallest backward distance is always selected for replacement. For the temporally-clustered reference pattern, the LRU replacement policy, which replaces the block with the largest backward distance, is used. Finally, for the probabilistic reference pattern, the LFU replacement policy that replaces the block with the lowest reference frequency is used.

3 Implementation of the DEAR Scheme in FreeBSD

Figure 3 shows the overall structure of the buffer cache manager for the DEAR scheme as implemented in FreeBSD 2.2.5. The DEAR scheme applies different replacement policies for different applications. This requires a split of the buffer cache management module into two parts, one for block allocation and the other for block replacement. The module responsible for block allocation is the System Cache Manager (SCM). There is one SCM in the system. The module responsible for block replacement is the Application Cache Manager (ACM). There is one ACM for each application. This organization is similar to that proposed for application-controlled file caching [12]. Both of the modules are located in the VFS (Virtual File System) layer and collaborate with each other for buffer allocation and block replacement.

An ACM is allocated to each process when the process is forked. When a block is referenced from the process, the associated ACM is called by the *bread()* or *bwrite()* procedure in the SCM (1) to locate the information about the referenced block using a hash table, (2) to update the block attribute that is changed by the current reference, (3) to place the block into a linked list that maintains the blocks referenced in the current detection period, and (4) to adjust the replacement order according to the application-specific replacement policy. To maintain the replacement order, the current implementation uses the linked list data structure for the LRU and MRU replacement policies and the heap data structure for the LFU replacement policy.

After the steps (1)-(4) are performed, a check is made to see whether the current detection period

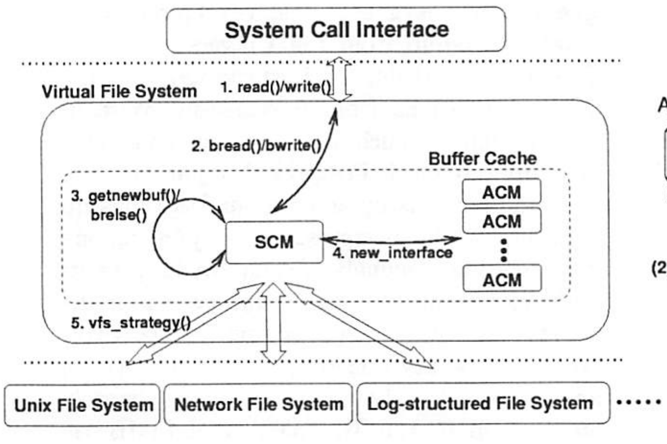


Figure 3: Overall structure of the DEAR scheme in FreeBSD 2.2.5.

is over. If so, the monitoring process explained in the previous section is invoked to detect the application's reference pattern. The detected reference pattern dictates the replacement policy of the ACM. If none of the detection conditions previously explained is satisfied, the default LRU replacement policy is used.

The structure of information maintained for each block by the ACM is <vnode #, block #, backward distance, frequency, forward distance, hp, bp, fp, cp>. The pointer hp is used to place the block into the hash table that is used to locate the information about the currently referenced block. The pointers bp and fp are used to place the block into the ordered lists for the backward distance and frequency block attribute types, respectively, which are constructed when the monitoring process is invoked. Finally, the pointer cp is used to place the block into the list of blocks referenced in the current detection period. This data structure is the main space overhead of the DEAR scheme.

The main time overhead of the DEAR scheme is that needed to order the blocks according to each block attribute value, which has an $O(n \log n)$ time complexity where n is the number of distinct blocks referenced in the detection period. This operation is invoked once at the end of each detection period for each block attribute type. Other time overheads include those needed to calculate the forward distance, backward distance, and frequency of blocks at the end of each detection period, which has a time complexity of $O(n)$ where n is the number of distinct blocks referenced in the detection period.

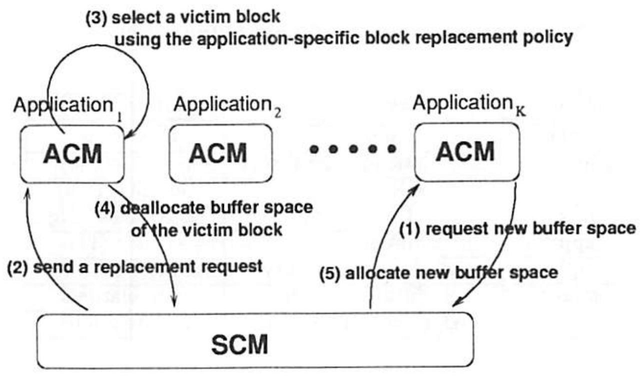

(3) select a victim block using the application-specific block replacement policy
Application 1
Application 2
Application K
ACM
ACM
ACM
(4) deallocate buffer space of the victim block
(1) request new buffer space
(2) send a replacement request
(5) allocate new buffer space
SCM

Figure 4: Interaction between ACM and SCM.

The ACM and SCM interact with each other as depicted in Figure 4. When an application misses in the buffer cache, the ACM for the application makes a request to the SCM for additional buffer space (step (1) in Figure 4). If the SCM does not have any free buffer space, it sends a replacement request to one of the ACMs (step (2)). This operation is performed in the *getnewbuf()* procedure in the SCM, and the selected ACM is the one associated with an application whose current reference pattern is sequential. If there is no such application, the SCM simply chooses the ACM of the application with the global LRU block. The selected ACM decides the victim block to be replaced using its current replacement policy (step (3)) and deallocates its space to the SCM (step (4)). The SCM allocates this space to the ACM that requested the space (step (5)).

4 Performance Evaluation

In this section, we present the results of the performance evaluation of the DEAR scheme. We first describe the experimental setup. Then, we give the results of reference pattern detection followed by the performance measurement results for both single applications and multiple applications. We also give results from sensitivity analysis for different cache sizes and detection periods. Finally, we compare the performance of the DEAR scheme with that of application-controlled file caching [12] through trace-driven simulations with the same set of application traces used in [12].

Table 1: Characteristics of the applications.

Application	Description	Input data (MB)
cscope	C examination tool	C code (9)
glimpse	information retrieval tool	text files (50)
sort	UNIX sort utility	text files (4.5)
link	UNIX link editor	object files (2.5)
cpp	C preprocessor	C code (11)
gnuplot	GNU plotting utility	numeric data (8)
postgres1	relational DB system	two relations
postgres2	relational DB system	four relations

4.1 Experimental Setup

The experiments were conducted with FreeBSD 2.2.5 on a 166MHZ Intel Pentium PC with 64MB RAM and a 2.1GB Quantum Fireball hard disk. The applications we used are described below and are summarized in Table 1.

cscope Cscope is an interactive C-source examination tool. It creates an index file named *cscope.out* from C sources and answers interactive queries like searching C symbols or finding specific functions or identifiers. We used cscope on kernel sources of roughly 9MB in size and executed queries that search for five literals.

glimpse Glimpse is a text information retrieval utility. It builds indexes for words and allows fast searching. Text files of roughly 50MB in size were indexed resulting in about 5MB of indexes. A search was done for lines that contain the keywords *multithread, realtime, DSM, continuous media,* and *diskspace.*

sort Sort is a utility that sorts lines of text files. A 4.5MB text file was used as input, and this file was sorted numerically using the first field as the key.

link Link is the UNIX link-editor. We used this application to build the FreeBSD kernel from about 2.5MB of object files.

cpp Cpp is the GNU C-compatible compiler preprocessor. The kernel source was used as input with the size of header files and C-source files of about 1MB and 10MB, respectively.

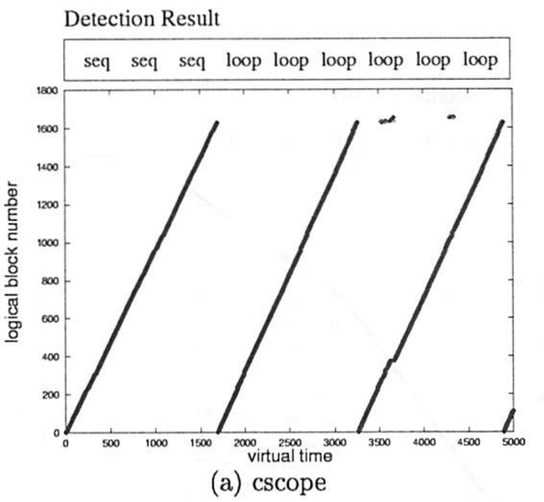
gnuplot Gnuplot is a command-line driven interactive plotting program. Using 8MB raw data, we plotted three-dimensional plots four times with different points of view.

postgres1 and postgres2 Postgres is a relational database system from the University of California at Berkeley. PostgresSQL version 6.2 and relations from a scaled-up Wisconsin benchmark such as thoustup and tenthoustup were used. Postgres1 is a join between the hundredthoustup and twohundredthoustup relations while postgres2 is a join among four relations, namely, fivehundredup, twothoustup, twentythoustup, and twohundredthoustup. The sizes of fivehundredup, twothoustup, twentythoustup, hundredthoustup, and twohundredthoustup are approximately 50KB, 150KB, 1.5MB, 7.5MB, and 15MB, respectively.

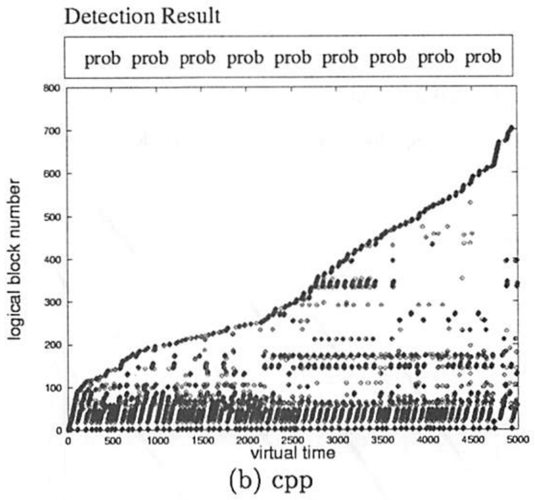
4.2 Detection Results

Figure 5 shows the results of the detection by the DEAR scheme for the cscope and cpp applications. In each graph, the x-axis is the virtual time and the y-axis is the logical block numbers of those referenced at the given time. The detection results are given at the top of the graph assuming a detection period of 500 references. For cscope, the DEAR scheme initially detects a sequential reference pattern but changes its detection to a looping reference pattern after the sequentially referenced blocks are re-accessed. This results from cscope always reading the file *cscope.out* sequentially whenever it receives a query about the C source. For cpp, the DEAR scheme detects a probabilistic reference pattern throughout the execution since as we can see from the graph, some blocks are more frequently accessed than others. This reference pattern results from the characteristic of cpp that header files are more frequently referenced than C files.

Figure 6 shows the detection results of the other applications. Although the result shows that the DEAR scheme performs reasonably well for the other applications, it also reveals the limitation of the current DEAR scheme, notably for the sort and postgres2 applications. They have either parallel or nested reference streams, which indicates a need for the proposed DEAR scheme to address more general reference patterns with arbitrary control structures such as parallel, sequence, and nested.



(a) cscope



(b) cpp

Figure 5: Block reference patterns and detection results for cscope and cpp.

4.3 Performance Measurements: Single Applications

We compared the performance of each application under the DEAR scheme with not only that under the LRU scheme in FreeBSD but also with those under the LFU and MRU schemes. For this purpose, we implemented the DEAR scheme as well as the LFU and MRU schemes in FreeBSD. We measured both the number of disk I/Os and the response time of each application for a 6MB buffer cache with block size set to 8KB. For the DEAR scheme, we set the length of the detection period to 500 and the number of sublists in the ordered lists to 5 for both the backward distance and frequency block attribute types. The performance of the DEAR scheme for different cache sizes, different detection periods, and different numbers of sublists in the ordered lists is discussed in Section 4.5.

Figure 7 shows the number of disk I/Os and the response time of the four schemes. The values reported here are the average of three separate executions and before each execution, the system was rebooted. From the results we observe the following:

- The DEAR scheme performs almost as good as the best of the other three schemes for all the applications we considered. Also, when compared with the LRU scheme in FreeBSD, the number of disk I/Os is reduced by up to 51% (for the cscope application) with an average of 23% and the response time by up to 35% (also for the cscope application) with an average of 12%.
- For the link application, there is no performance difference among the four schemes. This is because the input data to the link application is small (2.5MB), and thus all the blocks reside in the buffer cache after they are initially loaded.
- Postgres1 and postgres2 do not show as much improvement in the response time as that in the number of disk I/Os when using the DEAR scheme. This is because of the constant synchronization between the client (the psql utility that provides the user interface) and the server (the postgres process that performs the query processing and database management). For the gnuplot application, much time was spent for user mode computation and thus reduction in the number of disk I/Os also has a limited impact on the response time.
- Except for the above three applications, the ratio between the reduction in the number of disk I/Os and that in the response time is consistent. This indicates that the DEAR scheme incurs little extra overhead compared to those in the other schemes.

The last point is more evident in Figure 8 where the response time is divided into three components: I/O stall time, system time, and user time. For the LRU scheme of FreeBSD, the system time consists of VFS processing time, buffer cache manage-

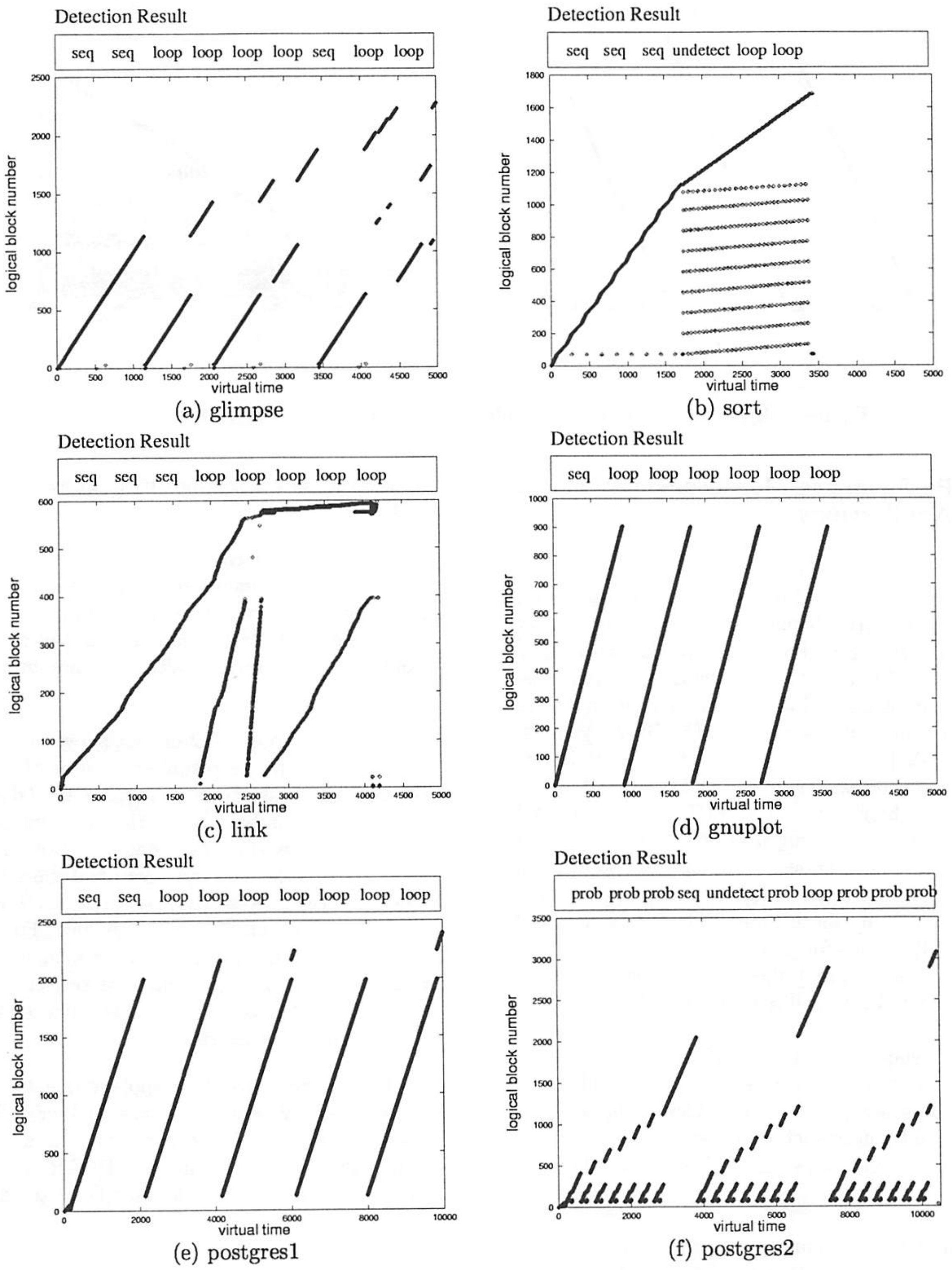
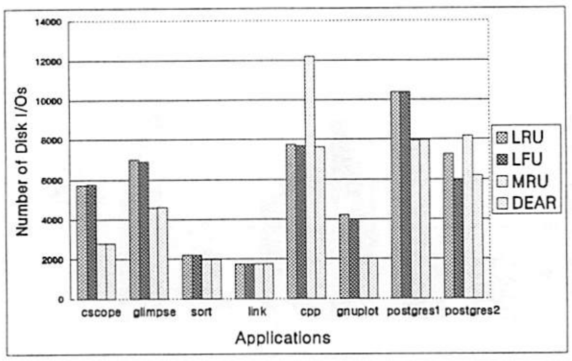
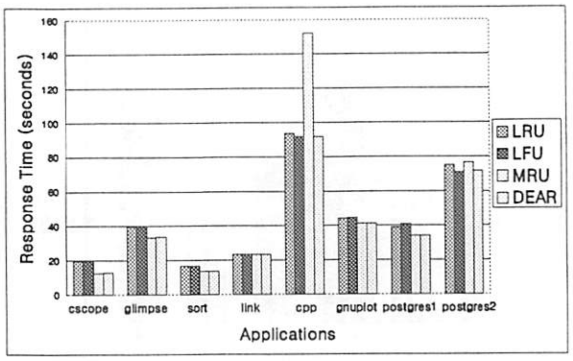


Figure 6: Block reference patterns and detection results for the other applications.



(a) Number of Disk I/Os



(b) Response Time

Figure 7: Single application performance.

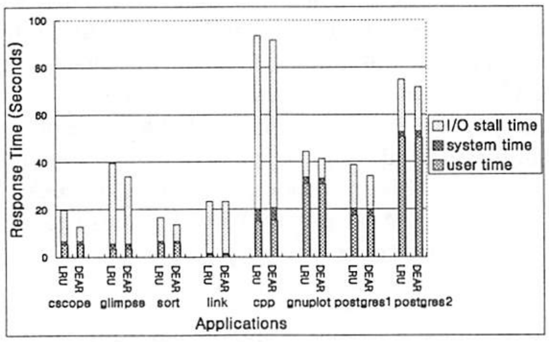


Figure 8: Decomposition of response time.

ment time, disk driver processing time, disk interrupt handling time and, data copy time from buffer cache to user space. On top of those, the DEAR scheme requires additional processing time such as those for sorting blocks according to block attribute values and maintaining block attribute values and forward distances. From Figure 8, we can notice that the system times of the two schemes are comparable meaning that the DEAR scheme incurs little additional overheads.

4.4 Performance Measurements: Multiple Applications

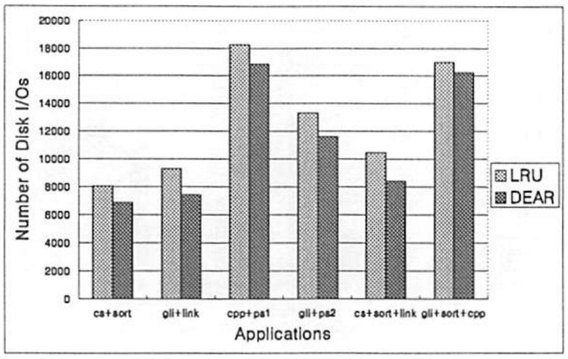
In real systems, multiple applications execute concurrently competing for limited buffer space. To test the DEAR scheme in such an environment, we ran several combinations of two or more of the applications with a buffer cache of 6MB and measured the total number of disk I/Os and the overall response time for both the DEAR scheme and the LRU scheme in FreeBSD. Again, we set the length of the detection period to 500 and the number of sublists in the ordered lists to 5.

The results in Figure 9 show that the number of disk I/Os is reduced by up to 20% (for the cscope+sort+link case) with an average of 12% and the overall response time by up to 18% (for the glimpse+link case) with an average of 8%.

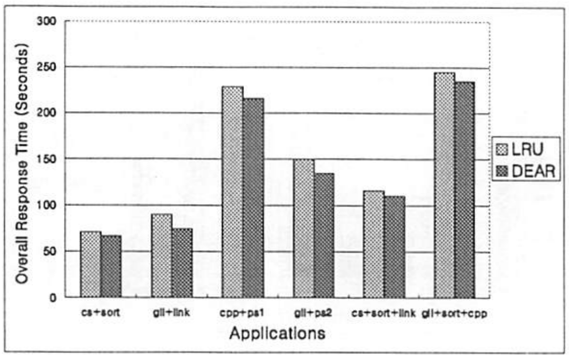
In the multiple application case, there are two possible benefits from using the proposed DEAR scheme. The first is from applying different replacement policies to different applications based on their detected reference patterns. The second is from giving preference to blocks that belong to an application with the sequential reference pattern when a replacement is needed. To quantify these benefits, we performed an experiment where even the LRU replacement policy gives preference to blocks belonging to an application with the sequential reference pattern, which we call the LRU-SEQ replacement policy.

Table 2: Performance comparison between the LRU-SEQ and the DEAR schemes.

Scheme	Response Time (seconds)			
	cs+sort	gli+link	cs+wc	gli+wc
LRU	70.96	89.87	81.27	89.97
LRU-SEQ	70.72	87.24	71.75	86.88
DEAR	66.61	74.29	62.88	82.36



(a) Number of Disk I/Os



(b) Overall response time

Figure 9: Multiple application performance.

Table 2 shows the results of the LRU-SEQ scheme for the 6MB buffer cache size. In the case of cscope+sort and glimpse+link, there is little difference between the LRU and the LRU-SEQ schemes, since the reference pattern of the four component applications is not sequential in the steady state. Wc is a utility that displays the numbers of lines, words, and characters in a file. Its steady state reference pattern is sequential. Replacing sort and link with wc, produces a significant difference in the response time between the LRU and the LRU-SEQ schemes. This results from the LRU-SEQ scheme allocating more buffer space to cscope (or glimpse) by replacing blocks of the wc application earlier than the usual LRU order. Still, there is a substantial difference in the response time between the LRU-SEQ scheme and the DEAR scheme indicating that the benefit from applying different replacement polices tailored for different applications is significant.

4.5 Sensitivity Analysis

4.5.1 Cache Size

Tables 3 and 4 compare the performance of the DEAR scheme against the LRU scheme for various buffer cache sizes for the single and multiple application cases, respectively. The results from the single application case show that as long as the total number of distinct blocks accessed by an application is greater than the number of blocks in the buffer cache, there is a substantial difference in the response time between the DEAR and the LRU schemes. However, when the number of distinct blocks of an application is smaller than the number of blocks in the buffer cache, all the blocks are cached in the buffer cache and the two schemes show similar performance. The latter behavior is most visible for the link application that has the smallest number of distinct blocks (about 310 blocks). For link, the DEAR and the LRU schemes provide similar response times.

Table 3: Single application performance for various buffer cache sizes.

Application	Scheme	Response Time (seconds)			
		2MB	4MB	6MB	8MB
cscope	DEAR	16.99	14.90	12.87	11.17
	LRU	19.79	19.79	19.77	19.77
glimpse	DEAR	39.12	35.68	33.73	32.87
	LRU	40.70	39.72	39.55	37.49
sort	DEAR	18.16	15.54	13.60	12.10
	LRU	18.50	17.45	16.59	14.68
link	DEAR	28.19	23.38	23.38	23.38
	LRU	29.65	23.35	23.35	23.35
cpp	DEAR	132.94	94.42	91.61	91.36
	LRU	159.59	97.94	93.39	91.82
gnuplot	DEAR	43.54	42.26	41.39	41.19
	LRU	44.30	44.30	44.30	44.30
postgres1	DEAR	38.37	36.16	34.22	32.17
	LRU	39.72	38.91	38.82	38.76
postgres2	DEAR	74.57	72.51	71.15	68.45
	LRU	82.93	74.93	74.75	73.93

For the multiple application case, the case where the total number of distinct blocks accessed by the component applications is smaller than the number

Table 5: The effect of the detection period on the performance of the DEAR scheme for the single application case.

Scheme	Detection Period	Response Time (seconds)						
		cscope	glimpse	sort	cpp	gnuplot	postgres1	postgres2
DEAR	100	12.85	33.70	13.72	98.81	40.92	34.62	76.56
	250	12.79	33.68	13.30	91.54	40.93	34.13	72.30
	500	12.87	33.73	13.60	91.61	41.39	34.22	71.15
	1000	13.52	36.26	13.88	91.78	41.66	34.53	72.41
	2000	15.20	36.45	15.77	91.99	42.36	34.84	72.53
LRU	N/A	19.77	39.55	16.59	93.39	44.30	38.82	74.75

Table 4: Multiple application performance for various buffer cache sizes.

Applications	Scheme	Response Time (seconds)			
		4MB	6MB	8MB	10MB
cs+sort	DEAR	70.4	66.6	62.9	53.5
	LRU	71.5	70.9	69.9	67.3
gli+link	DEAR	79.1	74.2	71.6	70.1
	LRU	94.5	89.8	79.1	77.9
cpp+ps1	DEAR	222.2	216.7	209.8	202.4
	LRU	236.5	229.9	226.6	226.1
gli+ps2	DEAR	145.6	139.8	132.5	128.3
	LRU	165.5	155.2	146.7	138.8
cs+sort+link	DEAR	116.1	112.7	106.7	101.8
	LRU	121.3	118.0	112.8	105.3
gli+sort+cpp	DEAR	245.9	235.5	215.8	207.2
	LRU	246.3	245.3	225.9	222.9

of blocks in the buffer cache does not occur and the DEAR scheme shows consistently better performance than the LRU scheme.

4.5.2 Detection Period and the number of Sublists

Determining the length of the detection period is an important design issue that requires a trade-off. If the detection period is too long, the scheme will not be adaptive to possible changes of the reference pattern within a detection period. On the other hand, if the period is too short, the scheme would incur too much overhead to be practical. Moreover, if the period is too short, a short burst of references may mislead the detection. For example, a probabilistic reference pattern may be mistaken for a looping reference pattern when a small number of blocks are repeatedly accessed over two detection periods while satisfying the detection condition for a looping reference pattern.

The above trade-off relationship is evident in Table 5 that gives the response time of all but the link application as the detection period varies from 100 to 2000. We exclude the link application since as we mentioned earlier all of its blocks fit into the buffer cache. Thus different detection periods do not make any difference. For most applications, the best performance was obtained when the detection period is either 250 or 500. The results also show that even with detection periods that are considerably smaller or larger than these optimal values, the DEAR scheme performs better than the LRU scheme in FreeBSD. The exceptions are with the cpp and postgres2 applications when the detection period is 100. In the two cases, the performance degradation is considerably larger than the others at the detection period of 100. A careful inspection of the results revealed that when the detection period is 100 the DEAR scheme mistakenly detects both applications to have a looping reference pattern when in reality it was part of a probabilistic reference pattern. The multiple application case shows a similar effect of the detection period on the performance as we can see in Table 6.

The number of sublists used in the detection process can also affect the detection results of the DEAR scheme. Table 7 gives the detection results of the DEAR scheme as the number of sublists increases from three to seven. From the results, we can notice that the number of sublists hardly affects the detection results although there is a slight increase in the number of undetected cases as the number of sublists increases due to a more strict detection rule. Remember that to detect a reference pattern the associated detection rule should be held for *all* the sublists.

Table 6: The effect of the detection period on the performance of the DEAR scheme for the multiple application case.

Scheme	Detection Period	Response Time (seconds)					
		cs+sort	gli+link	cpp+ps1	gli+ps2	cs+sort+link	gli+sort+cpp
DEAR	100	66.68	73.54	236.29	144.67	108.86	251.54
	250	65.84	73.41	216.62	136.86	108.62	230.61
	500	66.61	74.29	216.73	139.88	112.73	235.56
	1000	67.34	74.99	216.91	139.24	116.30	238.70
	2000	68.70	81.69	219.38	139.34	116.84	241.41
LRU	N/A	70.96	89.87	229.99	155.27	118.03	245.34

Table 7: The effect of the number of sublists on the detection results of the DEAR scheme.

Application	Detection Results		
	Number of sublists = 3	Number of sublists = 5	Number of sublists = 7
cscope	seq[3],loop[8]	seq[3],loop[8]	seq[3],loop[8]
glimpse	seq[4],loop[8]	seq[4],loop[8]	seq[3],loop[9]
sort	seq[3],loop[3]	seq[3],loop[2],undetect[1]	seq[3],loop[2],undetect[1]
link	seq[3],loop[5]	seq[3],loop[5]	seq[3],loop[5]
cpp	prob[18]	prob[18]	prob[13],undetect[5]
gnuplot	seq[1],loop[6]	seq[1],loop[6]	seq[1],loop[6]
postgres1	seq[5],loop[16]	seq[5],loop[16]	seq[5],loop[16]
postgres2	prob[13],loop[5],seq[2],undetect[1]	prob[12],loop[4],seq[2],undetect[3]	prob[11],loop[3],seq[2],undetect[5]

4.6 Comparison with Application-controlled File Caching

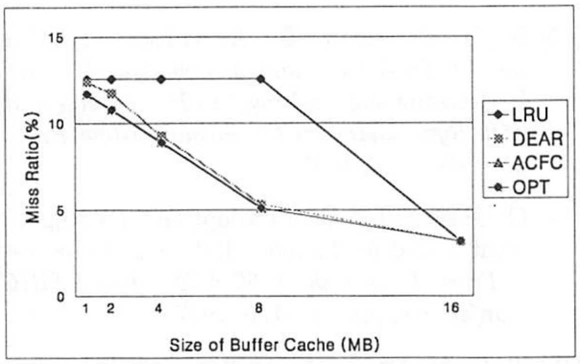
To compare the performance of the DEAR scheme with that of application-controlled file caching (ACFC) [12], we performed trace-driven simulations with the same set of three application traces used in [12]. The characteristics of the three applications and their traces can be found in [12]. Figure 10 shows the miss ratio of the three applications for the LRU, ACFC, DEAR, and OPT (off-line optimal) schemes when cache size increases from 1MB to 16MB. The results for the LRU, ACFC, and OPT schemes were borrowed from [12] and those for the DEAR scheme were obtained by simulating the DEAR scheme with detection period equal to 500 and the number of sublists in the ordered list equal to 5 for both backward distance and frequency block attribute types. The results show that the miss ratio of the DEAR scheme is comparable to that of the ACFC scheme, which utilizes user-level hints to guide the replacement decisions. The small difference between the two schemes results from the misses that occur before the DEAR scheme has a chance to detect the reference pattern.

5 Conclusions and Future Work

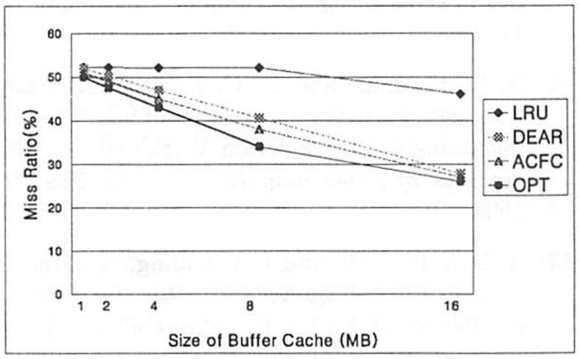
In this paper, we proposed a new buffer management scheme called DEAR (DEtection based Adaptive Replacement) that automatically detects the block reference pattern of applications as sequential, looping, temporally-clustered, or probabilistic without any user intervention. Based on the detected reference pattern, the proposed DEAR scheme applies an appropriate replacement policy to each application.

We implemented the DEAR scheme in FreeBSD 2.2.5 and measured its performance using several real applications. The results showed that compared with the buffer management scheme in FreeBSD the proposed scheme reduces the number of disk I/Os by up to 51% (with an average of 23%) and the response time by up to 35% (with an average of 12%) in the case of single application executions. For multiple applications, the reduction in the number of disk I/Os is by up to 20% (with an average of 12%) while the reduction in the overall response time is by up to 18% (with an average of 8%).

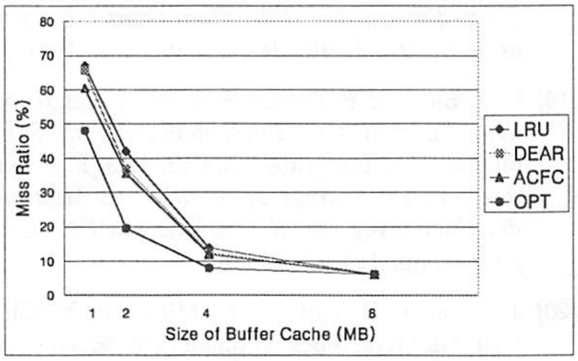
We also compared the performance of the DEAR scheme with that of application-controlled file caching [12] through trace-driven simulations with



(a) cscope



(b) linking kernel



(c) Postgres

Figure 10: Comparison with application-controlled file caching.

the same set of application traces used in [12]. The results showed that the DEAR scheme performs comparably to application-controlled file caching for the traces considered.

As we noted in Section 4.2, some applications have block reference behavior that cannot be characterized by a single reference pattern. One direction for future research is to extend the current DEAR scheme so that it can detect more complex reference patterns with parallel, sequence, and nested structures as well as to develop appropriate replacement policies for them. Another direction for future research is ,to study more advanced buffer allocation strategies for the DEAR scheme than the simple strategy explained in Section 3. A good buffer allocation strategy for the DEAR scheme should reward more to applications with larger reductions in the number of disk I/Os while preventing any one application from monopolizing the buffer space. Other directions for future research include applying the detection capability of the DEAR scheme to prefetching and considering block attribute types other than backward distance and frequency.

All the source code for the DEAR scheme and the applications run for the experiments can be found at `http://ssrnet.snu.ac.kr/~choijm/cache.html`.

Acknowledgments

We would like to thank Pei Cao, Edward W. Felten, and Kai Li for providing us with the traces we used in the experiments. We also would like to thank Yoonho Park, our shepherd, for his help in improving this paper and Donghee Lee for many helpful discussions. Finally, we would like to thank anonymous reviewers for their constructive comments and suggestions. This work was supported in part by the Korea Research Foundation for the program year of 1998.

References

[1] A. J. Smith, "Disk cache-miss ratio analysis and design considerations," *ACM Transactions on Computer Systems*, vol. 3, pp. 161–203, August 1985.

[2] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-Performance, Reliable Secondary Storage," *ACM Computing Surveys*, vol. 26, pp. 145–182, June 1994.

[3] J. T. Robinson and M. V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," in *Proceedings of the 1990 ACM SIGMETRICS Conference*, pp. 134–142, 1990.

[4] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," in *Proceedings of the 1993 ACM SIGMOD Conference*, pp. 297–306, 1993.

[5] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the Existence of a Spectrum of Policies that subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies." To appear in the Proceedings of the ACM SIGMETRICS'99, 1999.

[6] V. Phalke and B. Gopinath, "An Inter-Reference Gap Model for Temporal Locality in Program Behavior," in *Proceedings of the 1995 ACM SIGMETRICS Conference*, pp. 291–300, 1995.

[7] A. Dan and D. Sitaram, "A Generalized Interval Caching Policy for Mixed Interactive and Long Video Workloads," in *Proceedings of Multimedia Computing and Networking(MMCN) Conference*, pp. 344–351, 1996.

[8] C. Faloutsos, R. Ng, and T. Sellis, "Flexible and Adaptable Buffer Management Techniques for Database Management Systems," *IEEE Transactions on Computers*, vol. 44, pp. 546–560, April 1995.

[9] K. M. Curewitz, P. Krishnan, and J. S. Vitter, "Practical Prefetching via Data Compression," in *Proceedings of the 1993 ACM SIGMOD Conference*, pp. 257–266, 1993.

[10] J. Griffioen and R. Appleton, "Reducing File System Latency using a Predictive Approach," in *Proceedings of the 1994 Summer USENIX Conference*, pp. 197–207, 1994.

[11] T. M. Kroeger and D. D. E. Long, "Predicting File System Action from Prior Events," in *Proceedings of the 1996 Annual USENIX Technical Conference*, pp. 319–328, 1996.

[12] P. Cao, E. W. Felten, and K. Li, "Application-Controlled File Caching Policies," in *Proceedings of the USENIX Summer 1994 Technical Conference*, pp. 171–182, 1994.

[13] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," in *Proceedings of the 15th Symposium on Operating System Principles*, pp. 1–16, 1995.

[14] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," in *Proceedings of the 1997 ACM SIGMETRICS Conference*, pp. 115–126, 1997.

[15] T. C. Mowry, A. K. Demke, and O. Krieger, "Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications," in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pp. 3–17, 1996.

[16] B. K. Pasquale and G. C. Polyzos, "A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload," in *Proceedings of Supercomputing '93*, pp. 388–397, 1993.

[17] A. Dan, P. S. Yu, and J.-Y. Chung, "Characterization of Database Access Pattern for Analytic Prediction of Buffer Hit Probability," *VLDB Journal*, vol. 4, pp. 127–154, January 1995.

[18] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a Distributed File System," in *Proceedings of the 13th Symposium on Operating System Principles*, pp. 198–212, 1991.

[19] P. J. Shenoy, P. Goyal, S. S. Rao, and H. M. Vin, "Design and Implementation of Symphony: An Integrated Multimedia File System," in *Proceedings of ACM/SPIE Multimedia Computing and Networking(MMCN) Conference*, pp. 124–138, 1998.

[20] J. Choi, S. H. Noh, S. L. Min, and Y. Cho, "An Adaptive Block Management Scheme Using On-Line Detection of Block Reference Patterns," in *Proceeding of the Fourth IEEE International Workshop on Multi-Media Database Management Systems*, pp. 172–179, 1998.

[21] E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, New Jersey: Prentice-Hall, 1973.

A scalable and explicit event delivery mechanism for UNIX

Gaurav Banga gaurav@netapp.com
Network Appliance Inc., 2770 San Tomas Expressway, Santa Clara, CA 95051
Jeffrey C. Mogul mogul@pa.dec.com
Compaq Computer Corp. Western Research Lab., 250 University Ave., Palo Alto, CA, 94301
Peter Druschel druschel@cs.rice.edu
Department of Computer Science, Rice University, Houston, TX, 77005

Abstract

UNIX applications not wishing to block when doing I/O often use the *select()* system call, to wait for events on multiple file descriptors. The *select()* mechanism works well for small-scale applications, but scales poorly as the number of file descriptors increases. Many modern applications, such as Internet servers, use hundreds or thousands of file descriptors, and suffer greatly from the poor scalability of *select()*. Previous work has shown that while the traditional implementation of *select()* can be improved, the poor scalability is inherent in the design. We present a new event-delivery mechanism, which allows the application to register interest in one or more sources of events, and to efficiently dequeue new events. We show that this mechanism, which requires only minor changes to applications, performs independently of the number of file descriptors.

1 Introduction

An application must often manage large numbers of file descriptors, representing network connections, disk files, and other devices. Inherent in the use of a file descriptor is the possibility of delay. A thread that invokes a blocking I/O call on one file descriptor, such as the UNIX *read()* or *write()* systems calls, risks ignoring all of its other descriptors while it is blocked waiting for data (or for output buffer space).

UNIX supports non-blocking operation for *read()* and *write()*, but a naive use of this mechanism, in which the application polls each file descriptor to see if it might be usable, leads to excessive overheads.

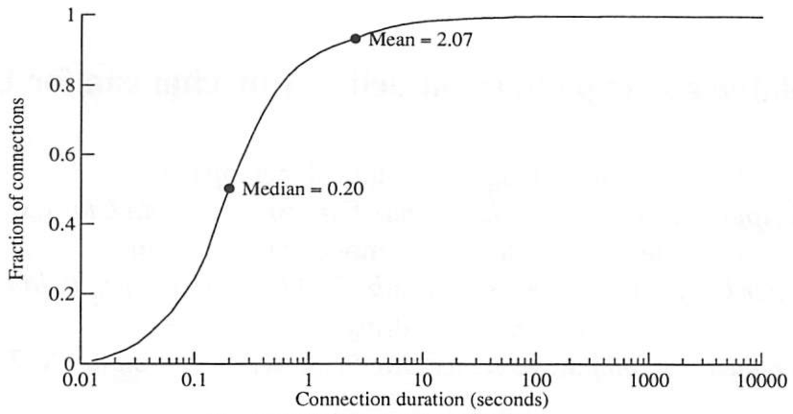
Alternatively, one might allocate a single thread to each activity, allowing one activity to block on I/O without affecting the progress of others. Experience with UNIX and similar systems has shown that this scales badly as the number of threads increases, because of the costs of thread scheduling, context-switching, and thread-state storage space[6, 9]. The use of a single process per connection is even more costly.

The most efficient approach is therefore to allocate a moderate number of threads, corresponding to the amount of available parallelism (for example, one per CPU), and to use non-blocking I/O in conjunction with an efficient mechanism for deciding which descriptors are ready for processing[17]. We focus on the design of this mechanism, and in particular on its efficiency as the number of file descriptors grows very large.

Early computer applications seldom managed many file descriptors. UNIX, for example, originally supported at most 15 descriptors per process[14]. However, the growth of large client-server applications such as database servers, and especially Internet servers, has led to much larger descriptor sets.

Consider, for example, a Web server on the Internet. Typical HTTP mean connection durations have been measured in the range of 2-4 seconds[8, 13]; Figure 1 shows the distribution of HTTP connection durations measured at one of Compaq's firewall proxy servers. Internet connections last so long because of long round-trip times (RTTs), frequent packet loss, and often because of slow (modem-speed) links used for downloading large images or binaries. On the other hand, modern single-CPU servers can handle about 3000 HTTP requests per second[19], and multiprocessors considerably more (albeit in carefully controlled environments). Queueing theory shows that an Internet Web server handling 3000 connections per second, with a mean duration of 2 seconds, will have about 6000 open connections to manage at once (assuming constant interarrival time).

In a previous paper[4], we showed that the BSD UNIX event-notification mechanism, the *select()* system call, scales poorly with increasing connection count. We showed that large connection counts do indeed occur in actual servers, and that the traditional implementation of *select()* could be improved significantly. However, we also found that even our improved *select()* implementation accounts for an unacceptably large share of the overall CPU time. This implies that, no matter how carefully it is implemented, *select()* scales poorly. (Some UNIX systems use a different system call, *poll()*, but we believe that this call has scaling properties at least as bad as those of *select()*, if not worse.)



N = 10,139,681 HTTP connections
Data from 21 October 1998 through 27 October 1998
Fig. 1: Cumulative distribution of proxy connection durations

The key problem with the *select()* interface is that it requires the application to inform the kernel, on each call, of the entire set of "interesting" file descriptors: i.e., those for which the application wants to check readiness. For each event, this causes effort and data motion proportional to the number of interesting file descriptors. Since the number of file descriptors is normally proportional to the event rate, the total cost of *select()* activity scales roughly with the square of the event rate.

In this paper, we explain the distinction between state-based mechanisms, such as *select()*, which check the current status of numerous descriptors, and event-based mechanisms, which deliver explicit event notifications. We present a new UNIX event-based API (application programming interface) that an application may use, instead of *select()*, to wait for events on file descriptors. The API allows an application to register its interest in a file descriptor once (rather than every time it waits for events). When an event occurs on one of these interesting file descriptors, the kernel places a notification on a queue, and the API allows the application to efficiently dequeue event notifications.

We will show that this new interface is simple, easily implemented, and performs independently of the number of file descriptors. For example, with 2000 connections, our API improves maximum throughput by 28%.

2 The problem with *select()*

We begin by reviewing the design and implementation of the *select()* API. The system call is declared as:

```
int select(
        int nfds,
        fd_set *readfds,
        fd_set *writefds,
        fd_set *exceptfds,
        struct timeval *timeout);
```

An *fd_set* is simply a bitmap; the maximum size (in bits) of these bitmaps is the largest legal file descriptor value, which is a system-specific parameter. The *readfds*, *writefds*, and *exceptfds* are in-out arguments, respectively corresponding to the sets of file descriptors that are "interesting" for reading, writing, and exceptional conditions. A given file descriptor might be in more than one of these sets. The *nfds* argument gives the largest bitmap index actually used. The *timeout* argument controls whether, and how soon, *select()* will return if no file descriptors become ready.

Before *select()* is called, the application creates one or more of the *readfds*, *writefds*, or *exceptfds* bitmaps, by asserting bits corresponding to the set of interesting file descriptors. On its return, *select()* overwrites these bitmaps with new values, corresponding to subsets of the input sets, indicating which file descriptors are available for I/O. A member of the *readfds* set is available if there is any available input data; a member of *writefds* is considered writable if the available buffer space exceeds a system-specific parameter (usually 2048 bytes, for TCP sockets). The application then scans the result bitmaps to discover the readable or writable file descriptors, and normally invokes handlers for those descriptors.

Figure 2 is an oversimplified example of how an application typically uses *select()*. One of us has shown[15] that the programming style used here is quite inefficient for large numbers of file descriptors, independent of the problems with *select()*. For example, the construction of the input bitmaps (lines 8 through 12 of Figure 2) should not be done explicitly before each call to *select()*; instead, the application should maintain shadow copies of the input bitmaps, and simply copy these shadows to *readfds* and *writefds*. Also, the scan of the result bitmaps, which are usually quite sparse, is best done word-by-word, rather than bit-by-bit.

Once one has eliminated these inefficiencies, however, *select()* is still quite costly. Part of this cost comes from the use of bitmaps, which must be created, copied into the kernel, scanned by the kernel, subsetted, copied out

```
fd_set readfds, writefds;
struct timeval timeout;
int i, numready;

timeout.tv_sec = 1; timeout.tv_usec = 0;

while (TRUE) {
  FD_ZERO(&readfds); FD_ZERO(&writefds);
  for (i = 0; i <= maxfd; i++) {
    if (WantToReadFD(i)) FD_SET(i, &readfds);
    if (WantToWriteFD(i)) FD_SET(i, &writefds);
  }
  numready = select(maxfd, &readfds,
                    &writefds, NULL, &timeout);
  if (numready < 1) {
    DoTimeoutProcessing();
      continue;
  }

  for (i = 0; i <= maxfd; i++) {
    if (FD_ISSET(i, &readfds)) InvokeReadHandler(i);
    if (FD_ISSET(i, &writefds)) InvokeWriteHandler(i);
  }
}
```

Fig. 2: Simplified example of how *select()* is used

of the kernel, and then scanned by the application. These costs clearly increase with the number of descriptors.

Other aspects of the *select()* implementation also scale poorly. Wright and Stevens provide a detailed discussion of the 4.4BSD implementation[23]; we limit ourselves to a sketch. In the traditional implementation, *select()* starts by checking, for each descriptor present in the input bitmaps, whether that descriptor is already available for I/O. If none are available, then *select()* blocks. Later, when a protocol processing (or file system) module's state changes to make a descriptor readable or writable, that module awakens the blocked process.

In the traditional implementation, the awakened process has no idea which descriptor has just become readable or writable, so it must repeat its initial scan. This is unfortunate, because the protocol module certainly knew what socket or file had changed state, but this information is not preserved. In our previous work on improving *select()* performance[4], we showed that it was fairly easy to preserve this information, and thereby improve the performance of *select()* in the blocking case.

We also showed that one could avoid most of the initial scan by remembering which descriptors had previously been interesting to the calling process (i.e., had been in the input bitmap of a previous *select()* call), and scanning those descriptors only if their state had changed in the interim. The implementation of this technique is somewhat more complex, and depends on set-manipulation operations whose costs are inherently dependent on the number of descriptors.

In our previous work, we tested our modifications using the Digital UNIX V4.0B operating system, and version 1.1.20 of the Squid proxy software[5, 18]. After doing our best to improve the kernel's implementation of *select()*, and Squid's implementation of the procedure that invokes *select()*, we measured the system's performance on a busy non-caching proxy, connected to the Internet and handling over 2.5 million requests/day.

We found that we had approximately doubled the system's efficiency (expressed as CPU time per request), but *select()* still accounted for almost 25% of the total CPU time. Table 1 shows a profile, made with the DCPI [1] tools, of both kernel and user-mode CPU activity during a typical hour of high-load operation.

In the profile *comm_select()*, the user-mode procedure that creates the input bitmaps for *select()* and that scans its output bitmaps, takes only 0.54% of the non-idle CPU time. Some of the 2.85% attributed to *memCopy()* and *memSet()* should also be charged to the creation of the input bitmaps (because the modified Squid uses the shadow-copy method). (The profile also shows a lot of time spent in *malloc()*-related procedures; a future version of Squid will use pre-allocated pools to avoid the overhead of too many calls to *malloc()* and *free()*[22].)

However, the bulk of the *select()*-related overhead is in the kernel code, and accounts for about two thirds of the total non-idle kernel-mode CPU time. Moreover, this measurement reflects a *select()* implementation that we had already improved about as much as we thought possible. Finally, our implementation could not avoid costs dependent on the number of descriptors, implying that the *select()*-related overhead scales worse than linearly. Yet these costs did not seem to be related to intrinsically useful work. We decided to design a scalable replace-

CPU %	Non-idle CPU %	Procedure	Mode
65.43%	100.00%	all non-idle time	kernel
34.57%		all idle time	kernel
16.02%	24.49%	all select functions	kernel
9.42%	14.40%	*select*	kernel
3.71%	5.67%	*new_soo_select*	kernel
2.82%	4.31%	*new_selscan_one*	kernel
0.03%	0.04%	*new_undo_scan*	kernel
15.45%	23.61%	*malloc*-related code	user
4.10%	6.27%	*in_pcblookup*	kernel
2.88%	4.40%	all TCP functions	kernel
0.94%	1.44%	*memCopy*	user
0.92%	1.41%	*memset*	user
0.88%	1.35%	*bcopy*	kernel
0.84%	1.28%	*read_io_port*	kernel
0.72%	1.10%	*_doprnt*	user
0.36%	0.54%	*comm_select*	user

Profile on 1998-09-09 from 11:00 to 12:00 PDT
mean load = 56 requests/sec.
peak load ca. 131 requests/sec

Table 1: Profile - modified kernel, Squid on live proxy

ment for *select()*.

2.1 The *poll()* system call

In the System V UNIX environment, applications use the *poll()* system call instead of *select()*. This call is declared as:

```
struct pollfd {
        int      fd;
        short    events;
        short    revents;
};

int poll(
        struct pollfd filedes[];
        unsigned int nfds;
        int timeout    /* in milliseconds */);
```

The *filedes* argument is an in-out array with one element for each file descriptor of interest; *nfds* gives the array length. On input, the *events* field of each element tells the kernel which of a set of conditions are of interest for the associated file descriptor *fd*. On return, the *revents* field shows what subset of those conditions hold true. These fields represent a somewhat broader set of conditions than the three bitmaps used by *select()*.

The *poll()* API appears to have two advantages over *select()*: its array compactly represents only the file descriptors of interest, and it does not destroy the input fields of its in-out argument. However, the former advantage is probably illusory, since *select()* only copies 3 bits per file descriptor, while *poll()* copies 64 bits. If the number of interesting descriptors exceeds 3/64 of the highest-numbered active file descriptor, *poll()* does more copying than *select()*. In any event, it shares the same scaling problem, doing work proportional to the number of interesting descriptors rather than constant effort, per event.

3 Event-based vs. state-based notification mechanisms

Recall that we wish to provide an application with an efficient and scalable means to decide which of its file descriptors are ready for processing. We can approach this in either of two ways:

1. A *state-based* view, in which the kernel informs the application of the current state of a file descriptor (e.g., whether there is any data currently available for reading).

2. An *event-based* view, in which the kernel informs the application of the occurrence of a meaningful event for a file descriptor (e.g., whether new data has been added to a socket's input buffer).

The *select()* mechanism follows the state-based approach. For example, if *select()* says a descriptor is ready for reading, then there is data in its input buffer. If the application reads just a portion of this data, and then calls *select()* again before more data arrives, *select()* will again report that the descriptor is ready for reading.

The state-based approach inherently requires the kernel to check, on every notification-wait call, the status of each member of the set of descriptors whose state is being tested. As in our improved implementation of *select()*, one can elide part of this overhead by watching for events that change the state of a descriptor from unready to ready. The kernel need not repeatedly re-test the state of a descriptor known to be unready.

However, once *select()* has told the application that a descriptor is ready, the application might or might not perform operations to reverse this state-change. For example, it might not read anything at all from a ready-for-reading input descriptor, or it might not read all of the pending data. Therefore, once *select()* has reported that a descriptor is ready, it cannot simply ignore that descriptor on future calls. It must test that descriptor's state, at least until it becomes unready, even if no further I/O events occur. Note that elements of *writefds* are usually ready.

Although *select()* follows the state-based approach, the kernel's I/O subsystems deal with events: data packets arrive, acknowledgements arrive, disk blocks arrive, etc. Therefore, the *select()* implementation must transform notifications from an internal event-based view to an external state-based view. But the "event-driven" applications that use *select()* to obtain notifications ultimately follow the event-based view, and thus spend effort tranforming information back from the state-based model. These dual transformations create extra work.

Our new API follows the event-based approach. In this model, the kernel simply reports a stream of events to the application. These events are monotonic, in the sense that they never decrease the amount of readable data (or writable buffer space) for a descriptor. Therefore, once an event has arrived for a descriptor, the application can either process the descriptor immediately, or make note of the event and defer the processing. The kernel does not track the readiness of any descriptor, so it does not perform work proportional to the number of descriptors; it only performs work proportional to the number of events.

Pure event-based APIs have two problems:

1. Frequent event arrivals can create excessive communication overhead, especially for an application that is not interested in seeing every individual event.
2. If the API promises to deliver information about each individual event, it must allocate storage proportional to the event rate.

Our API does not deliver events asynchronously (as would a signal-based mechanism; see Section 8.2), which helps to eliminate the first problem. Instead, the API allows an application to efficiently discover descriptors that have had event arrivals. Once an event has arrived for a descriptor, the kernel coalesces subsequent event arrivals for that descriptor until the application learns of the first one; this reduces the communication rate, and avoids the need to store per-event information. We believe that most applications do not need explicit per-event information, beyond that available in-band in the data stream.

By simplifying the semantics of the API (compared to *select()*), we remove the necessity to maintain information in the kernel that might not be of interest to the application. We also remove a pair of transformations between the event-based and state-based views. This improves the scalability of the kernel implementation, and leaves the application sufficient flexibility to implement the appropriate event-management algorithms.

4 Details of the programming interface

An application might not be always interested in events arriving on all of its open file descriptors. For example, as mentioned in Section 8.1, the Squid proxy server temporarily ignores data arriving in dribbles; it would rather process large buffers, if possible.

Therefore, our API includes a system call allowing a thread to declare its interest (or lack of interest) in a file descriptor:

```
#define     EVENT_READ        0x1
#define     EVENT_WRITE       0x2
#define     EVENT_EXCEPT      0x4

int declare_interest(int fd,
                     int interestmask,
                     int *statemask);
```

The thread calls this procedure with the file descriptor in question. The *interestmask* indicate whether or not the thread is interested in reading from or writing to the descriptor, or in exception events. If *interestmask* is zero, then the thread is no longer interested in any events for the descriptor. Closing a descriptor implicitly removes any declared interest.

Once the thread has declared its interest, the kernel tracks event arrivals for the descriptor. Each arrival is added to a per-thread queue. If multiple threads are interested in a descriptor, a per-socket option selects between two ways to choose the proper queue (or queues). The default is to enqueue an event-arrival record for each interested thread, but by setting the SO_WAKEUP_ONE flag, the application indicates that it wants an event arrival delivered only to the first eligible thread.

If the *statemask* argument is non-NULL, then *declare_interest()* also reports the current state of the file descriptor. For example, if the EVENT_READ bit is set in this value, then the descriptor is ready for reading. This feature avoids a race in which a state change occurs after the file has been opened (perhaps via an *accept()* system call) but before *declare_interest()* has been called. The implementation guarantees that the *statemask* value reflects the descriptor's state before any events are added to the thread's queue. Otherwise, to avoid missing any events, the application would have to perform a non-blocking *read* or *write* after calling *declare_interest()*.

To wait for additional events, a thread invokes another new system call:

```
typedef struct {
  int fd;
  unsigned mask;
} event_descr_t;

int get_next_event(int array_max,
                   event_descr_t *ev_array,
                   struct timeval *timeout);
```

The *ev_array* argument is a pointer to an array, of length *array_max*, of values of type *event_descr_t*. If any events are pending for the thread, the kernel dequeues, in FIFO order, up to *array_max* events[1]. It reports these dequeued events in the *ev_array* result array. The *mask* bits in each *event_descr_t* record, with the same definitions as used in *declare_interest()*, indicate the current

[1] A FIFO ordering is not intrinsic to the design. In another paper[3], we describe a new kernel mechanism, called *resource containers*, which allows an application to specify the priority in which the kernel enqueues events.

state of the corresponding descriptor *fd*. The function return value gives the number of events actually reported.

By allowing an application to request an arbitrary number of event reports in one call, it can amortize the cost of this call over multiple events. However, if at least one event is queued when the call is made, it returns immediately; we do not block the thread simply to fill up its *ev_array*.

If no events are queued for the thread, then the call blocks until at least one event arrives, or until the timeout expires.

Note that in a multi-threaded application (or in an application where the same socket or file is simultaneously open via several descriptors), a race could make the descriptor unready before the application reads the *mask* bits. The application should use non-blocking operations to read or write these descriptors, even if they appear to be ready. The implementation of *get_next_event()* does attempt to try to report the current state of a descriptor, rather than simply reporting the most recent state transition, and internally suppresses any reports that are no longer meaningful; this should reduce the frequency of such races.

The implementation also attempts to coalesce multiple reports for the same descriptor. This may be of value when, for example, a bulk data transfer arrives as a series of small packets. The application might consume all of the buffered data in one system call; it would be inefficient if the application had to consume dozens of queued event notifications corresponding to one large buffered read. However, it is not possible to entirely eliminate duplicate notifications, because of races between new event arrivals and the *read*, *write*, or similar system calls.

5 Use of the programming interface

Figure 3 shows a highly simplified example of how one might use the new API to write parts of an event-driven server. We omit important details such as error-handling, multi-threading, and many procedure definitions.

The *main_loop()* procedure is the central event dispatcher. Each iteration starts by attempting to dequeue a batch of events (here, up to 64 per batch), using *get_next_event()* at line 9. If the system call times out, the application does its timeout-related processing. Otherwise, it loops over the batch of events, and dispatches event handlers for each event. At line 16, there is a special case for the socket(s) on which the application is listening for new connections, which is handled differently from data-carrying sockets.

We show only one handler, for these special listen-sockets. In initialization code not shown here, these listen-sockets have been set to use the non-blocking option. Therefore, the *accept()* call at line 30 will never block, even if a race with the *get_next_event()* call somehow causes this code to run too often. (For example, a remote client might close a new connection before we have a chance to accept it.) If *accept()* does successfully return the socket for a new connection, line 31 sets it to use non-blocking I/O. At line 32, *declare_interest()* tells the kernel that the application wants to know about future read and write events. Line 34 tests to see if any data became available before we called *declare_interest()*; if so, we read it immediately.

6 Implementation

We implemented our new API by modifying Digital UNIX V4.0D. We started with our improved *select()* implementation [4], reusing some data structures and support functions from that effort. This also allows us to measure our new API against the best known *select()* implementation without varying anything else. Our current implementation works only for sockets, but could be extended to other descriptor types. (References below to the "protocol stack" would then include file system and device driver code.)

For the new API, we added about 650 lines of code. The *get_next_event()* call required about 320 lines, *declare_interest()* required 150, and the remainder covers changes to protocol code and support functions. In contrast, our previous modifications to *select()* added about 1200 lines, of which we reused about 100 lines in implementing the new API.

For each application thread, our code maintains four data structures. These include INTERESTED.read, INTERESTED.write, and INTERESTED.except, the sets of descriptors designated via *declare_interest()* as "interesting" for reading, writing, and exceptions, respectively. The other is HINTS, a FIFO queue of events posted by the protocol stack for the thread.

A thread's first call to *declare_interest()* causes creation of its INTERESTED sets; the sets are resized as necessary when descriptors are added. The HINTS queue is created upon thread creation. All four sets are destroyed when the thread exits. When a descriptor is closed, it is automatically removed from all relevant INTERESTED sets.

Figure 4 shows the kernel data structures for an example in which a thread has declared read interest in descriptors 1 and 4, and write interest in descriptor 0. The three INTERESTED sets are shown here as one-byte bitmaps, because the thread has not declared interest in any higher-numbered descriptors. In this example, the HINTS queue for the thread records three pending events, one each for descriptors 1, 0, and 4.

A call to *declare_interest()* also adds an element to the corresponding socket's "reverse-mapping" list; this element includes both a pointer to the thread and the descriptor's index number. Figure 5 shows the kernel

```
#define MAX_EVENTS 64
struct event_descr_t event_array[MAX_EVENTS];

main_loop(struct timeval timeout)
{
  int i, n;

  while (TRUE) {
    n = get_next_event(MAX_EVENTS, &event_array, &timeout);
    if (n < 1) {
      DoTimeoutProcessing(); continue;
    }

    for (i = 0; i < n; i++) {
      if (event_array[i].mask & EVENT_READ)
        if (ListeningOn(event_array[i].fd))
          InvokeAcceptHandler(event_array[i].fd);
        else
          InvokeReadHandler(event_array[i].fd);
      if (event_array[i].mask & EVENT_WRITE)
        InvokeWriteHandler(event_array[i].fd);
    }
  }
}

InvokeAcceptHandler(int listenfd)
{
  int newfd, statemask;

  while ((newfd = accept(listenfd, NULL, NULL)) >= 0) {
    SetNonblocking(newfd);
    declare_interest(newfd, EVENT_READ|EVENT_WRITE,
                     &statemask);
    if (statemask & EVENT_READ)
      InvokeReadHandler(newfd);
  }
}
```

Fig. 3: Simplified example of how the new API might be used

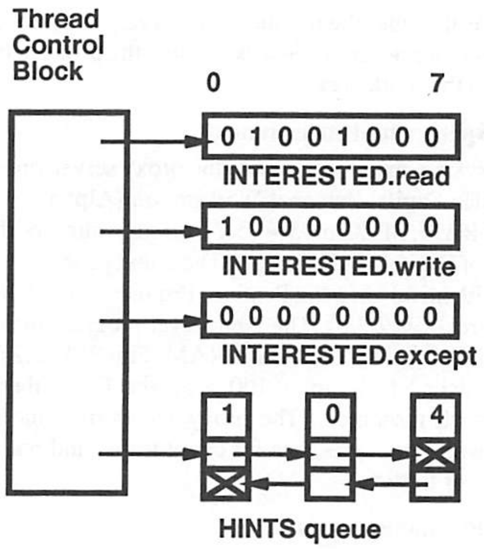


Fig. 4: Per-thread data structures

data structures for an example in which Process 1 and Process 2 hold references to Socket A via file descriptors 2 and 4, respectively. Two threads of Process 1 and one thread of Process 2 are interested in Socket A, so the reverse-mapping list associated with the socket has pointers to all three threads.

When the protocol code processes an event (such as data arrival) for a socket, it checks the reverse-mapping list. For each thread on the list, if the index number is found in the thread's relevant INTERESTED set, then a notification element is added to the thread's HINTS queue.

To avoid the overhead of adding and deleting the reverse-mapping lists too often, we never remove a reverse-mapping item until the descriptor is closed. This means that the list is updated at most once per descriptor lifetime. It does add some slight per-event overhead for a socket while a thread has revoked its interest in that descriptor; we believe this is negligible.

We attempt to coalesce multiple event notifications for a single descriptor. We use another per-thread bitmap, in-

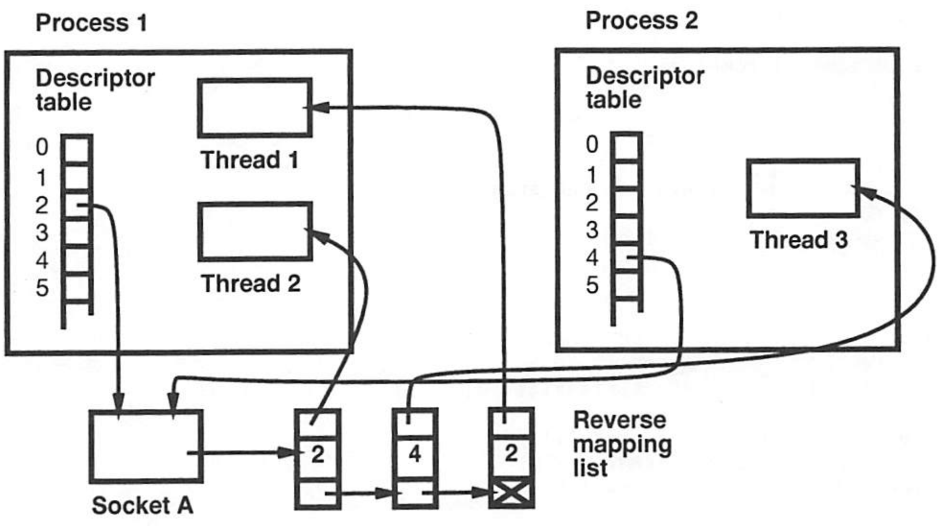


Fig. 5: Per-socket data structures

dexed by file descriptor number, to note that the HINTS queue contains a pending element for the descriptor. The protocol code tests and sets these bitmap entries; they are cleared once *get_next_event()* has delivered the corresponding notification. Thus, N events on a socket between calls to *get_next_event()* lead to just one notification.

Each call to *get_next_event()*, unless it times out, dequeues one or more notification elements from the HINTS queue in FIFO order. However, the HINTS queue has a size limit; if it overflows, we discard it and deliver events in descriptor order, using a linear search of the INTERESTED sets – we would rather deliver things in the wrong order than block progress. This policy could lead to starvation, if the *array_max* parameter to *get_next_event()* is less than the number of descriptors, and may need revision.

We note that there are other possible implementations for the new API. For example, one of the anonymous reviewers suggested using a linked list for the per-thread queue of pending events, reserving space for one list element in each socket data structure. This approach seems to have several advantages when the SO_WAKEUP_ONE option is set, but might not be feasible when each event is delivered to multiple threads.

7 Performance

We measured the performance of our new API using a simple event-driven HTTP proxy program. This proxy does not cache responses. It can be configured to use either *select()* or our new event API.

In all of the experiments presented here, we generate load using two kinds of clients. The "hot" connections come from a set of processes running the S-Client software [2], designed to generate realistic request loads, characteristic of WAN clients. As in our earlier work [4], we also use a *load-adding client* to generate a large number of "cold" connections: long-duration dummy connections that simulate the effect of large WAN delays. The load-adding client process opens as many as several thousand connections, but does not actually send any requests. In essence, we simulate a load with a given arrival rate and duration distribution by breaking it into two pieces: S-Clients for the arrival rate, and load-adding clients for the duration distribution.

The proxy relays all requests to a Web server, a single-process event-driven program derived from thttpd [20], with numerous performance improvements. (This is an early version of the Flash Web server [17].) We take care to ensure that the clients, the Web server, and the network itself are never bottlenecks. Thus, the proxy server system is the bottleneck.

7.1 Experimental environment

The system under test, where the proxy server runs, is a 500MHz Digital Personal Workstation (Alpha 21164, 128MB RAM, SPECint95 = 15.7), running our modified version of Digital UNIX V4.0D. The client processes run on four identical 166Mhz Pentium Pro machines (64MB RAM, FreeBSD 2.2.6). The Web server program runs on a 300 MHz Pentium II (128MB RAM, FreeBSD 2.2.6).

A switched full-duplex 100 Mbit/sec Fast Ethernet connects all machines. The proxy server machine has two network interfaces, one for client traffic and one for Web-server traffic.

7.2 API function costs

We performed experiments to find the basic costs of our new API calls, measuring how these costs scale with the number of connections per process. Ideally, the costs should be both low and constant.

In these tests, S-Client software simulates HTTP clients generating requests to the proxy. Concurrently, a load-adding client establishes some number of cold connections to the proxy server. We started measurements only after a dummy run warmed the Web server's file cache. During these measurements, the proxy's CPU is saturated, and the proxy application never blocks in *get_next_event()*; there are always events queued for delivery.

The proxy application uses the Alpha's cycle counter to measure the elapsed time spent in each system call; we report the time averaged over 10,000 calls.

To measure the cost of *get_next_event()*, we used S-Clients generating requests for a 40 MByte file, thus causing thousands of events per connection. We ran trials with *array_max* (the maximum number of events delivered per call) varying between 1 and 10; we also varied the number of S-Client processes. Figure 6 shows that the cost per call, with 750 cold connections, varies linearly with *array_max*, up to a point limited (apparently) by the concurrency of the S-Clients.

For a given *array_max* value, we found that varying the number of cold connections between 0 and 2000 has almost no effect on the cost of *get_next_event()*, accounting for variation of at most 0.005% over this range.

We also found that increasing the hot-connection rate did not appear to increase the per-event cost of *get_next_event()*. In fact, the event-batching mechanism reduces the per-event cost, as the proxy falls further behind. The cost of all event API operations in our implementation is independent of the event rate, as long as the maximum size of the HINTS queue is configured large enough to hold one entry for each descriptor of the process.

To measure the cost of the *declare_interest()* system call, we used 32 S-Clients making requests for a 1 KByte file. We made separate measurements for the "declaring interest" case (adding a new descriptor to an INTERESTED set) and the "revoking interest" case (removing a descriptor); the former case has a longer code path. Figure 7 shows slight cost variations with changes in the number of cold connections, but these may be measurement artifacts.

7.3 Proxy server performance

We then measured the actual performance of our simple proxy server, using either *select()* or our new API. In these experiments, all requests are for the same (static) 1 Kbyte file, which is therefore always cached in the Web server's memory. (We ran additional tests using 8 Kbyte files; space does not permit showing the results, but they display analogous behavior.)

In the first series of tests, we always used 32 hot connections, but varied the number of cold connections between 0 and 2000. The hot-connection S-Clients are configured to generate requests as fast as the proxy system can handle; thus we saturated the proxy, but never overloaded it. Figure 8 plots the throughput achieved for three kernel configurations: (1) the "classical" implementation of *select()*, (2) our improved implementation of *select()*, and (3) the new API described in this paper. All kernels use a scalable version of the *ufalloc()* file-descriptor allocation function [4]; the normal version does not scale well. The results clearly indicate that our new API performs independently of the number of cold connections, while *select()* does not. (We also found that the proxy's throughput is independent of *array_max*.)

In the second series of tests, we fixed the number of cold connections at 750, and measured response time (as seen by the clients). Figure 9 shows the results. When using our new API, the proxy system exhibits much lower latency, and saturates at a somewhat higher request load (1348 requests/sec., vs. 1291 request/sec. for the improved *select()* implementation).

Table 2 shows DCPI profiles of the proxy server in the three kernel configurations. These profiles were made using 750 cold connections, 50 hot connections, and a total load of 400 requests/sec. They show that the new event API significantly increases the amount of CPU idle time, by almost eliminating the event-notification overhead. While the classical *select()* implementation consumes 34% of the CPU, and our improved *select()* implementation consumes 12%, the new API consumes less than 1% of the CPU.

8 Related work

To place our work in context, we survey other investigations into the scalability of event-management APIs, and the design of event-management APIs in other operating systems.

8.1 Event support in NetBIOS and Win32

The NetBIOS interface[12] allows an application to wait for incoming data on multiple network connections. NetBIOS does not provide a procedure-call interface; instead, an application creates a "Network Control Block" (NCB), loads its address into specific registers, and then invokes NetBIOS via a software interrupt. NetBIOS provides a command's result via a callback.

The NetBIOS "receive any" command returns (calls back) when data arrives on any network "session" (connection). This allows an application to wait for arriving data on an arbitrary number of sessions, without having to enumerate the set of sessions. It does not appear possible to wait for received data on a subset of the active sessions.

The "receive any" command has numerous limitations, some of which are the result of a non-extensible design. The NCB format allows at most 254 sessions, which obviates the need for a highly-scalable implement-

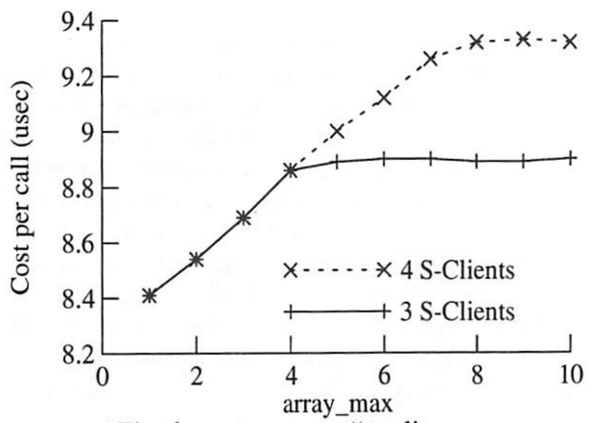


Fig. 6: *get_next_event()* scaling

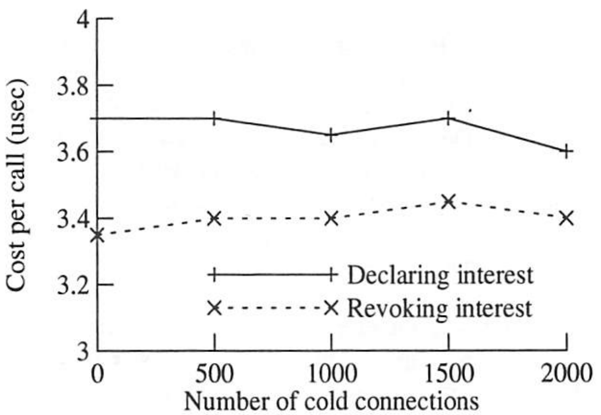


Fig. 7: *declare_interest()* scaling

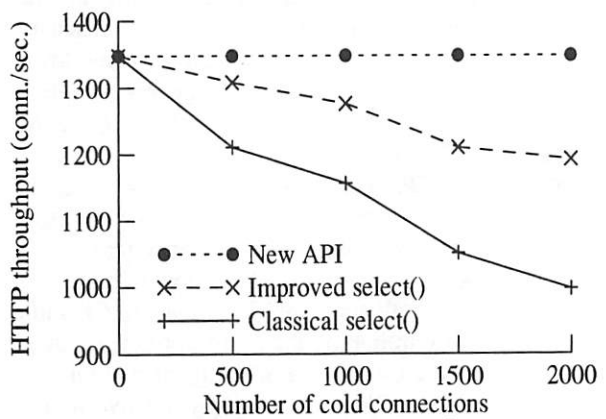


Fig. 8: HTTP rate vs. cold connections

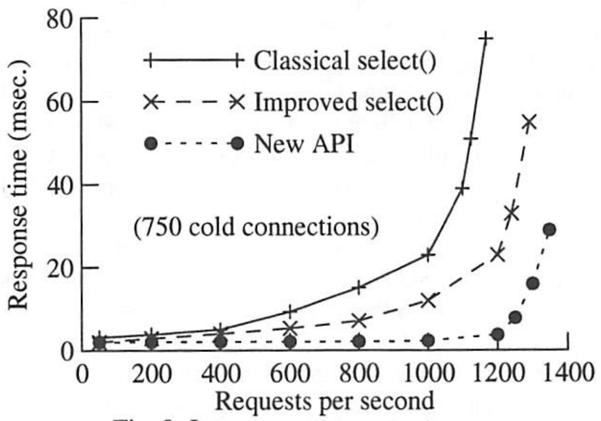


Fig. 9: Latency vs. request rate

ation. The command does not allow an application to discover when a once-full output buffer becomes writable, nor does it apply to disk files.

In the Win32 programming environment[10], the use of NetBIOS is strongly discouraged. Win32 includes a procedure named *WaitForMultipleObjects()*, declared as:

```
DWORD WaitForMultipleObjects(
  DWORD cObjects,
    // number of handles in handle array
  CONST HANDLE * lphObjects,
    // address of object-handle array
  BOOL fWaitAll,
    // flag: wait for all or for just one
  DWORD dwTimeout
    // time-out interval in milliseconds
  );
```

This procedure takes an array of Win32 objects (which could include I/O handles, threads, processes, mutexes, etc.) and waits for either one or all of them to complete. If the *fWaitAll* flag is FALSE, then the returned value is the array index of the ready object. It is not possible to learn about multiple objects in one call, unless the application is willing to wait for completion on all of the listed objects.

This procedure, like *select()*, might not scale very well to large numbers of file handles, for a similar reason: it passes information about all potential event sources every time it is called. (In any case, the object-handle array may contain no more than 64 elements.) Also, since *WaitForMultipleObjects* must be called repeatedly to obtain multiple events, and the array is searched linearly, a frequent event rate on objects early in the array can starve service for higher-indexed objects.

Windows NT 3.5 added a more advanced mechanism for detecting I/O events, called an I/O completion port (IOCP)[10, 21]. This ties together the threads mechanism with the I/O mechanism. An application calls *CreateIoCompletionPort()* to create an IOCP, and then makes an additional call to *CreateIoCompletionPort()* to associate each interesting file handle with that IOCP. Each such call also provides an application-specified "CompletionKey" value that will be associated with the file handle.

An application thread waits for I/O completion events using the *GetQueuedCompletionStatus()* call:

```
BOOL GetQueuedCompletionStatus(
  HANDLE CompletionPort,
  LPDWORD lpNumberOfBytesTransferred,
  LPDWORD CompletionKey,
  LPOVERLAPPED *lpOverlapped,
  DWORD dwMillisecondTimeout);
```

Upon return, the *CompletionKey* variable holds the

Classical select() CPU %	Scalable select() CPU %	New event API CPU %	Procedure	Mode
18.09%	33.01%	59.01%	all idle time	kernel
33.51%	*12.02%*	*0.68%*	*all kernel* select *or event functions*	kernel
13.78%	N.A.	N.A.	*soo_select()*	kernel
9.11%	N.A.	N.A.	*selscan()*	kernel
8.40%	N.A.	N.A.	*undo_scan()*	kernel
2.22%	12.02%	N.A.	*select()*	kernel
N.A.	0.57%	N.A.	*new_soo_select()*	kernel
N.A.	0.47%	N.A.	*new_selscan_one()*	kernel
N.A.	N.A.	0.40%	*get_next_event()*	kernel
N.A.	N.A.	0.15%	*declare_interest()*	kernel
N.A.	N.A.	0.13%	*revoke_interest()*	kernel
2.01%	1.95%	1.71%	*_Xsyscall()*	kernel
1.98%	1.88%	1.21%	*main()*	user
1.91%	1.90%	1.69%	*_doprnt()*	user
1.63%	1.58%	1.54%	*memset()*	user
1.29%	1.31%	1.47%	*read_io_port()*	kernel
1.11%	1.15%	1.20%	*syscall()*	kernel
1.09%	1.11%	1.11%	*_XentInt()*	kernel
1.08%	1.06%	1.19%	*malloc()*	kernel

750 cold connections, 50 hot connections, 400 requests/second, 1KB/request

Table 2: Effect of event API on system CPU profile

value associated, via *CreateIoCompletionPort()*, with the corresponding file handle. Several threads might be blocked in this procedure waiting for completion events on the same IOCP. The kernel delivers the I/O events in FIFO order, but selects among the blocked threads in LIFO order, to reduce context-switching overhead.

The IOCP mechanism seems to have no inherent limits on scaling to large numbers of file descriptors or threads. We know of no experimental results confirming its scalability, however.

Once a handle has been associated with an IOCP, there is no way to disassociate it, except by closing the handle. This somewhat complicates the programmer's task; for example, it is unsafe to use as the CompletionKey the address of a data structure that might be reallocated when a file handle is closed. Instead, the application should use a nonce value, implying another level of indirection to obtain the necessary pointer. And while the application might use several IOCPs to segregate file handles into different priority classes, it cannot move a file handle from one IOCP to another as a way of adjusting its priority.

Some applications, such as the Squid proxy[5, 18], temporarily ignore I/O events on an active file descriptor, to avoid servicing data arriving as a lengthy series of small dribbles. This is easily done with the UNIX *select()* call, by removing that descriptor from the input bitmap; it is not clear if this can be done using an IOCP.

Hu et al.[11] discuss several different NT event dispatching and concurrency models in the context of a Web server, and show how the server's performance varies according to the model chosen. However, they did not measure how performance scales with large numbers of open connections, but limited their measurements to at most 16 concurrent clients.

In summary, the IOCP mechanism in Windows NT is similar to the API we propose for UNIX, and predates our design by several years (although we were initially unaware of it). The differences between the designs may or may not be significant; we look forward to a careful analysis of IOCP performance scaling. Our contribution is not the concept of a pending-event queue, but rather its application to UNIX, and our quantitative analysis of its scalability.

8.2 Queued I/O completion signals in POSIX

The POSIX[16] API allows an application to request the delivery of a signal (software interrupt) when I/O is possible for a given file descriptor. The POSIX Realtime Signals Extension allows an application to request that delivered signals be queued, and that the signal handler be invoked with a parameter giving the associated file descriptor. The combination of these facilities provides a scalable notification mechanism.

We see three problems that discourage the use of signals. First, signal delivery is more expensive than the specialized event mechanism we propose. On our test system, signal delivery (for SIGIO) requires 10.7 usec, versus about 8.4 usec for *get_next_event()* (see figure 6), and (unlike *get_next_event()*) the signal mechanism cannot batch notifications for multiple descriptors in one invocation. Second, asynchronous invocation of handlers implies the use of some sort of locking mechanism, which adds overhead and complexity. Finally, the use of signals eliminates application control over which thread is invoked.

8.3 Port sets in Mach

The Mach operating system[24] depends on message-based communication, using "ports" to represent message end-points. Ports are protected with capability-like "send rights" and "receive rights." All system operations are performed using messages; for example, virtual memory faults are converted into messages sent to the backing-store port of the associated memory object. Other communication models, such as TCP byte streams, are constructed on top of this message-passing layer.

Each port has a queue of pending messages. A thread may use the *msg_receive()* system call to retrieve a message from the queue of a single port, or wait for a message to arrive if the queue is empty.

A thread with receive rights for many ports may create a "port set", a first-class object containing an arbitrary subset of these receive rights[7]. The thread may then invoke *msg_receive()* on that port set (rather than on the underlying ports), receiving messages from all of the contained ports in FIFO order. Each message is marked with the identity of the original receiving port, allowing the thread to demultiplex the messages. The port set approach scales efficiently: the time required to retrieve a message from a port set should be independent of the number of ports in that set.

Port sets are appropriate for a model in which all communication is done with messages, and in which the system provides the necessary facilities to manage message ports (not necessarily a simple problem[7]). Introducing port sets into UNIX, where most communication follows a byte-stream model, might require major changes to applications and existing components.

9 Future work

The *select()* mechanism can be confusing in multithreaded programs, especially on multiprocessors. Because *select()* returns the state of a descriptor, instead of an event notification, two threads blocked in *select()* could awaken at the same time, and would need additional synchronization to avoid handling the same descriptor. Our event-based API should make writing threaded applications more natural, because (with the SO_WAKEUP_ONE option described in Section 4) it delivers each event at most once. We have not yet explored this area in detail.

Our existing API requires each thread in a process to call *declare_interest()* for each descriptor that it is interested in. This requirement might add excessive overhead for a multi-threaded program using a large pool of interchangeable worker threads. We could augment the API with another system call:

```
int declare_processwide_interest(int fd,
                                 int interestmask,
                                 int *statemask);
```

The result of this system call would be the equivalent of invoking *declare_interest()* in every existing and future thread of the calling process. (It might also implicitly set SO_WAKEUP_ONE for the descriptor.) After this call, any thread of the process could wait for events on this descriptor using *get_next_event()*.

An application handling thousands of descriptors might want to set event-delivery priorities, to control the order in which the kernel delivers events. In another paper [3], we introduced the *resource container* abstraction, which (among other benefits) allows an application to set kernel-level priorities for descriptor processing. In that paper we showed how an event-based API, such as the one presented here, is a useful component of end-to-end priority control in networked applications. We look forward to gaining experience with the combination of priority control and an event-based API in complex applications.

10 Summary

We showed that the scalability of an operating system's event notification mechanism has a direct effect on application performance scalability. We also showed that the *select()* API has inherently poor scalability, but that it can be replaced with a simple event-oriented API. We implemented this API and showed that it does indeed improve performance on a real application.

11 Acknowledgments

We would like to thank Mitch Lichtenberg for helping us understand the NetBIOS, Win32, and Windows NT event-management APIs. We would also like to thank the anonymous reviewers for their suggestions.

This work was done while Gaurav Banga was a student at Rice University. It was supported in part by NSF Grants CCR-9803673 and CCR-9503098, by Texas TATP Grant 003604, and by an equipment loan by the Digital Equipment Corporation subsidiary of Compaq Computer Corporation.

References

[1] J. Anderson, L. M. Berc, et al. Continuous profiling: Where have all the cycles gone? In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 1–14, San Malo, France, Oct. 1997.

[2] G. Banga and P. Druschel. Measuring the Capacity of a Web Server. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems*, pages 61–71, Monterey, CA, Dec. 1997.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. 3rd. Symp. on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, February 1999.

[4] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proc. 1998 USENIX Annual Technical Conf.*, pages 1–12, New Orleans, LA, June 1998. USENIX.

[5] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 USENIX Technical Conference*, pages 153–163, San Diego, CA, Jan. 1996.

[6] J. B. Chen. Personal communication, Nov. 1998.

[7] R. P. Draves. A Revised IPC Interface. In *Proc. USENIX Mach Conference*, pages 101–122, Burlington, VT, October 1990.

[8] S. D. Gribble and E. A. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proc. USENIX Symposium on Internet Technologies and Systems*, pages 207–218, Monterey, CA, December 1997.

[9] D. Grunwald. Personal communication, Mar. 1998.

[10] J. M. Hart. *Win32 System Programming*. Addison Wesley Longman, Reading, MA, 1997.

[11] J. Hu, I. Pyarali, and D. C. Schmidt. Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks. In *Proc. Global Internet Conference (held as part of GLOBECOM '97)*, Phoenix, AZ, November 1997.

[12] IBM. *Local Area Network Technical Reference*. Research Triangle Park, NC, 4th edition, 1990.

[13] T. M. Kroeger, D. D. E. Long, and J. C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proc. Symposium on Internet Technologies and Systems*, pages 13–22, Monterey, CA, December 1997. USENIX.

[14] J. Lions. *Lion's Commentary on UNIX 6th Edition with Source Code*. Peer-to-Peer Communications, San Jose, CA, 1996.

[15] J. C. Mogul. Speedier Squid: A case study of an Internet server performance problem. *;login:*, pages 50–58, February 1999.

[16] The Open Group, Woburn, MA. *The Single UNIX Specification, Version 2 - 6 Vol Set for UNIX 98*, 1997.

[17] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proc. 1999 USENIX Annual Technical Conf.*, Monterey, CA, June 1998. USENIX.

[18] Squid. http://squid.nlanr.net/Squid/.

[19] The Standard Performance Evaluation Corporation (SPEC). SPECweb96 Results. http://www.specbench.org/osg/web96/results/.

[20] thttpd. http://www.acme.com/software/thttpd/.

[21] J. Vert. Writing Scalable Applications for Windows NT. http://www.microsoft.com/win32dev/base/SCALABIL.HTM, 1995.

[22] D. Wessels. Personal communication, September 1998.

[23] G. Wright and W. Stevens. *TCP/IP Illustrated Volume 2*. Addison-Wesley, Reading, MA, 1995.

[24] M. Young, A. Tevanian, Jr., R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proc. 11th Symposium on Operating Systems Principles*, pages 63–76, Austin, TX, November 1987.

The Pebble Component-Based Operating System

Eran Gabber, Christopher Small, John Bruno†, José Brustoloni and Avi Silberschatz

Information Sciences Research Center
Lucent Technologies—Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974
{eran, chris, jbruno, jcb, avi}@research.bell-labs.com

†*Also affiliated with the University of California at Santa Barbara*

Abstract

Pebble is a new operating system designed with the goals of flexibility, safety, and performance. Its architecture combines a set of features heretofore not found in a single system, including (a) a minimal privileged mode nucleus, responsible for switching between protection domains, (b) implementation of all system services by replaceable user-level components with minimal privileges (including the scheduler and all device drivers) that run in separate protection domains enforced by hardware memory protection, and (c) generation of code specialized for each possible cross-domain transfer. The combination of these techniques results in a system with extremely inexpensive cross-domain calls that makes it well-suited for both efficiently specializing the operating system on a per-application basis and supporting modern component-based applications.

1 Introduction

A new operating system project should address a real problem that is not currently being addressed; constructing yet another general purpose POSIX- or Windows32-compliant system that runs standard applications is not a worthwhile goal in and of itself. The Pebble operating system was designed with the goal of providing flexibility, safety, and high performance to applications in ways that are not addressed by standard desktop operating systems.

Flexibility is important for specialized systems, often referred to as embedded systems. The term is a misnomer, however, as embedded systems run not just on microcontrollers in cars and microwaves, but also on high-performance general purpose processors found in routers, laser printers, and hand-held computing devices.

Safety is important when living in today's world of mobile code and component-based applications. Although safe languages such as Java [Gosling96] and Limbo [Dorward97] can be used for many applications, hardware memory protection is important when code is written in unsafe languages such as C and C++.

High performance cannot be sacrificed to provide safety and flexibility. History has shown us that systems are chosen primarily for their performance characteristics; safety and flexibility almost always come in second place. Any system structure added to support flexibility and safety cannot come at a significant decrease in performance; if possible, a new system should offer better performance than existing systems.

Early in the project, the designers of Pebble decided that to maximize system flexibility Pebble would run as little code as possible in its privileged mode nucleus. If a piece of functionality could be run at user level, it was removed from the nucleus. This approach makes it easy to replace, layer, and offer alternative versions of operating system services.

Each user-level component runs in its own *protection domain*, isolated by means of hardware memory protection. All communication between protection domains is done by means of a generalization of interrupt handlers, termed *portals*. Only if a portal exists between protection domain A and protection domain B can A invoke a service offered by B. Because each protection domain has its own *portal table*, by restricting the set of portals available to a protection domain, threads in that domain are efficiently isolated from services to which they should not have access.

Portals are not only the basis for flexibility and safety in Pebble, they are also the key to its high performance. Specialized, tamper-proof code can be generated for each portal, using a simple interface definition language. Portal code can thus be optimized for its portal, saving and restoring the minimum necessary state, or encapsulating and compiling out demultiplexing decisions and run-time checks.

The remainder of this paper is structured as follows. In Section 2 we discuss related work. In Section 3 we describe the architecture of Pebble, and in Section 4 we discuss the portal mechanism and its uses in more detail. Section 5 covers several key implementation issues of Pebble. Section 6 introduces the idea of implementing a protected, application-transparent "sandbox" via portal interposition, and shows the performance overhead of such a sandbox. Section 7 compares the performance of Pebble and OpenBSD on our test hardware, a MIPS R5000 processor. Section 8 reviews the current status of Pebble and discusses our plans for future work. We summarize in Section 9, and include a short code example that implements the sandbox discussed in Section 6.

2 Related Work

Pebble has the same general structure as classical microkernel operating systems such as Mach [Acetta86], Chorus [Rozer88], and Windows NT [Custer92], consisting of a privileged mode kernel and a collection of user level servers. Pebble's protected mode nucleus is much smaller and has fewer responsibilities than the kernels of these systems, and in that way is much more like the L4 microkernel [Liedtke95]. L4 and Pebble share a common philosophy of running as little code in privileged mode as possible. Where L4 implements IPC and minimal virtual memory management in privileged mode, Pebble's nucleus includes only code to transfer threads from one protection domain to another and a small number of support functions that require kernel mode.

Mach provides a facility to intercept system calls and service them at user level [Golub90]. Pebble's portal mechanism, which was designed for high-performance cross-protection-domain transfer, can be used in a similar way, taking an existing application component and interposing one or more components between the application component and the services it uses.

Pebble's architecture is closer in spirit to the nested process architecture of Fluke [Ford96]. Fluke provides an architecture in which virtual operating systems can be layered, with each layer only affecting the performance of the subset of the operating system interface it implements. For example, the presence of multiple virtual memory management "nesters" (*e.g.*, to provide demand paging, distributed shared memory, and persistence) would have no effect on the cost of invoking file system operations such as `read` and `write`. The Fluke model requires that system functionality be replaced in groups; a memory management nester must implement all of the functions in the virtual memory interface specification. Pebble portals can be replaced piecemeal, which permits finer-grained extensibility.

The Exokernel model [Engler95, Kaashoek97] attempts to "exterminate all OS abstractions," with the privileged mode kernel in charge of protecting resources, but leaving resource abstraction to user level application code. As with the Exokernel approach, Pebble moves the implementation of resource abstractions to user level, but unlike the Exokernel, Pebble provides a set of abstractions, implemented by user-level operating system components. Pebble OS components can be added or replaced, allowing alternate OS abstractions to coexist or override the default set.

Pebble can use the interposition technique discussed in Section 6 to wrap a "sandbox" around untrusted code. Several extensible operating system projects have studied the use of software techniques, such as safe languages (*e.g.*, Spin [Bershad95]) and software fault isolation (*e.g.*, VINO [Seltzer96]), for this purpose. Where software techniques require faith in the safety of a compiler, interpreter, or software fault isolation tool, a sandbox implemented by portal interposition and hardware memory protection provides isolation at the hardware level, which may be simpler to verify than software techniques.

Philosophically, the Pebble approach to sandboxing is akin to that provided by the Plan 9 operating system [Pike90]. In Plan 9, nearly all resources are modeled as files, and each process has its own file name space. By restricting the namespace of a process, it can be effectively isolated from resources to which it should not have access. In contrast with Plan 9, Pebble can restrict access to any service, not just those represented by files.

Pebble applies techniques developed by Bershad et al. [Bershad89], Massalin [Massalin92], and Pu et al. [Pu95] to improve the performance of IPC. Bershad's results showed that IPC data size tends to be very small (which fits into registers) or large (which is passed by sharing memory pages). Massalin's work on the Synthesis project, and, more recently, work by Pu et al. on the Synthetix project, studied the use of generating specialized code to improve performance.

Pebble was inspired by the SPACE project [Probert91]. Many of the concepts and much of the terminology of the project come from Probert's work; *e.g.*, SPACE provided us with the idea of cross-domain communication as a generalization of interrupt handling.

The Spring kernel [Mitchell94] provided cross-protection domain calls via doors, which are similar to Pebble's portals. However, Spring's doors are used only for implementing operations on objects, and do not include general purpose parameter manipulations.

The Kea system [Veitch96] is very similar to Pebble. It provides protection domains, inter-domain calls via portals and portal remapping. However, Kea's portals do not perform general parameter manipulations like Pebble. Parameter manipulations, such as sharing memory pages, are essential for efficient communication between components.

The MMLite system [Helander98] is a component-based system that provides a wide selection of object-oriented components that are assembled into an application system. MMLite's components are space efficient. However, MMLite does not use any memory protection, and all components execute in the same protection domain.

Like Dijkstra's THE system [Dijkstra68], Pebble hides the details of interrupts from higher level components and uses only semaphores for synchronization.

Some CISC processors provide a single instruction that performs a full context switch. A notable example is the Intel x86 task switch via a call gate [Intel94]. However, this instruction takes more than 100 machine cycles.

3 Philosophy and Architecture

The Pebble philosophy consists of the following four key ideas.

The privileged-mode nucleus is as small as possible. If something can be run at user level, it is.

The privileged-mode nucleus is only responsible for switching between protection domains. In a perfect world, Pebble would include only one privileged-mode instruction, which would transfer control from one protection domain to the next. By minimizing the work done in privileged mode, we reduce both the amount of privileged code and the time needed to perform essential privileged mode services.

The operating system is built from fine-grained replaceable components, isolated through the use of hardware memory protection.

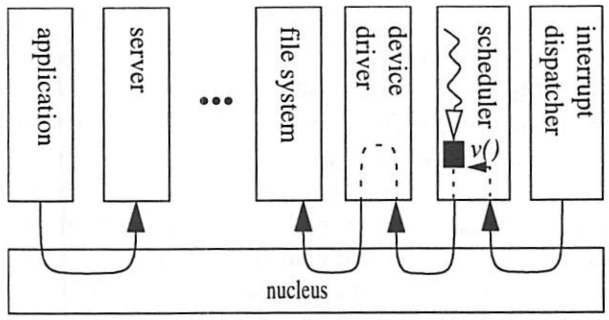


Figure 1. Pebble architecture. Arrows denote portal traversals. On the right, an interrupt causes a device driver's semaphore to be incremented, unblocking the device driver's thread (see Section).

The functionality of the operating system is implemented by trusted user-level components. The components can be replaced, augmented, or layered.

The architecture of Pebble is based around the availability of hardware memory protection; Pebble, as described here, requires a memory management unit.

The cost of transferring a thread from one protection domain to another should be small enough that there is no performance-related reason to co-locate services.

It has been demonstrated that the cost of using hardware memory protection on the Intel x86 can be made extremely small [Liedtke97], and we believe that if it can be done on the x86, it could be done anywhere. Our results bear us out—Pebble can perform a one-way IPC in 114 machine cycles on a MIPS R5000 processor (see Section 7 for details).

Transferring a thread between protection domains is done by a generalization of hardware interrupt handling, termed portal traversal. Portal code is generated dynamically and performs portal-specific actions.

Hardware interrupts, IPC, and the Pebble equivalent of system calls are all handled by the portal mechanism. Pebble generates specialized code for each portal to improve run-time efficiency. Portals are discussed in more detail in the following section.

3.1 Protection Domains, Portals and Threads

Each component runs in its own protection domain (*PD*). A protection domain consists of a set of pages, represented by a page table, and a set of portals, which are generalized interrupt handlers, stored in the protection domain's portal table. A protection domain may share both pages and portals with other protection domains. Figure 1 illustrates the Pebble architecture.

Portals are used to handle both hardware interrupts and software traps and exceptions. The existence of a portal from PD_A to PD_B means that a thread running in PD_A can invoke a specific entry point of PD_B (and then return). Associated with each portal is code to transfer a thread from the invoking domain to the invoked domain. Portal code copies arguments, changes stacks, and maps pages shared between the domains. Portal code is specific to its portal, which allows several important optimizations to be performed (described below).

Portals are usually generated in pairs. The call portal transfers control from domain PD_A to PD_B, and the return portal allows PD_B to return to PD_A. In the following discussion we will omit the return portal for brevity.

Portals are generated when certain resources are created (*e.g.* semaphores) and when clients connect to servers (*e.g.* when files are opened). Some portals are created at the system initialization time (*e.g.* interrupt and exception handling portals).

A scheduling priority, a stack, and a machine context are associated with each Pebble thread. When a thread traverses a portal, no scheduling decision is made; the thread continues to run, with the same priority, in the invoked protection domain. Once the thread executes in the invoked domain, it may access all of the resources available in the invoked domain, while it can no longer access the resources of the invoking domain. Several threads may execute in the same protection domain at the same time, which means that they share the same portal table and all other resources.

As part of a portal traversal, the portal code can manipulate the page tables of the invoking and/or invoked protection domains. This most commonly occurs when a thread wishes to map, for the duration of the IPC, a region of memory belonging to the invoking protection domain into the virtual address space of the invoked protection domain; this gives the thread a window into the address space of the invoking protection domain while running in the invoked protection domain. When the thread returns, the window is closed.

Such a memory window can be used to save the cost of copying data between protection domains. Variations include windows that remain open (to share pages between protection domains), windows that transfer pages from the invoking domain to the invoked domain (to implement tear-away write) and windows that transfer pages from the invoked domain to the invoker (to implement tear-away read).

Note that although the portal code may modify VM data structures, only the VM manager and the portal manager (which generates portal code) share the knowledge about these data structures. The Pebble nucleus itself is oblivious to those data structures.

3.2 Safety

Pebble implements a safe execution environment by a combination of hardware memory protection that prevents access to memory outside the protection domain, and by limiting the access to the domain's portal table. An protection domain may access only the portals it inherited from its parent and new portals that were generated on its behalf by the portal manager. The portal manager may restrict access to new portals in conjunction with the name server. A protection domain cannot transfer a portal it has in its portal table to an unrelated domain. Moreover, the parent domain may intercept all of its child portal calls, including calls that indirectly manipulate the child's portal table, as described in Section 6.

3.3 Server Components

As part of the Pebble philosophy, system services are provided by operating system server components, which run in user mode protection domains. Unlike applications, server components are trusted, so they may be granted limited privileges not afforded to application components. For example, the scheduler runs with interrupts disabled, device drivers have device registers mapped into their memory region, and the portal manager may add portals to protection domains (a protection domain cannot modify its portal table directly).

There are many advantages of implementing services at user level. First, from a software engineering standpoint, we are guaranteed that a server component will use only the exported interface of other components. Second, because each server component is only given the privileges that it needs to do its job, a programming error in one component will not directly affect other components. If a critical component fails (*e.g.*, VM) the system as a whole will be affected—but a bug in console device driver will not overwrite page tables.

Additionally, as user-level servers can be interrupted at any time, this approach has the possibility of offering lower interrupt latency time. Given that server components run at user level (including interrupt-driven threads), they can use blocking synchronization primitives, which simplifies their design. This is in contrast with handlers that run at interrupt level, which must not block, and require careful coding to synchronize with the upper parts of device drivers.

3.4 The Portal Manager

The Portal Manager is the operating system component responsible for instantiating and managing portals. It is privileged in that it is the only component that is permitted to modify portal tables.

Portal instantiation is a two-step process. First, the server (which can be a Pebble system component or an application component) registers the portal with the portal manager, specifying the entrypoint, the interface definition, and the name of the portal. Second, a client component requests that a portal with a given name be opened. The portal manager may call the name server to identify the portal and to verify that the client is permitted to open the portal. If the name server approves the access, the portal manger generates the code for the portal, and installs the portal in the client's portal table. The portal number of the newly generated portal is returned to the client. A client may also inherit a portal from its parent as the result of a `domain_fork()`, as described in Section 4.5.

To invoke the portal, a thread running in the client loads the portal number into a register and traps to the nucleus. The trap handler uses the portal number as an index into the portal table and jumps to the code associated with the portal. The portal code transfers the thread from the invoking protection domain to the invoked protection domain and returns to user level. As stated above, a portal transfer does not involve the scheduler in any way. (Section 5.4 describes the only exception to this rule.)

Portal interfaces are written using a (tiny) interface definition language, as described in Section 4.4. Each portal argument may be processed or transformed by portal code. The argument transformation may involve a function of the nucleus state, such as inserting the identity of the calling thread or the current time. The argument transformation may also involve other servers. For example, a portal argument may specify the address of a memory window to be mapped into the receiver's address space. This transformation requires the manipulation of data structures in the virtual memory server.

The design of the portal mechanism presents the following conflict: on one hand, in order to be efficient, the argument transformation code in the portal may need to have access to private data structures of a trusted server (*e.g.*, the virtual memory system); on the other hand, trusted servers should be allowed to keep their internal data representations private.

The solution we advocate is to allow trusted servers, such as the virtual memory manager, to register argument transformation code templates with the portal manager. (Portals registered by untrusted services would be required to use the standard argument types.) When the portal manager instantiates a portal that uses such an argument, the appropriate type-specific code is generated as part of the portal. This technique allows portal code to be both efficient (by inlining code that transforms arguments) and encapsulated (by allowing servers to keep their internal representations private). Although portal code that runs in kernel mode has access to server-specific data structures, these data structures cannot be accessed by other servers. The portal manager currently supports argument transformation code of a single trusted server, the virtual memory server.

3.5 Scheduling and Synchronization

Because inter-thread synchronization is intrinsically a scheduling activity, synchronization is managed entirely by the user-level scheduler. When a thread creates a semaphore, two portals (for *P* and *V*) are added to its portal table that transfer control to the scheduler. When a thread in the domain invokes *P*, the thread is transferred to the scheduler; if the *P* succeeds, the scheduler returns. If the *P* fails, the scheduler marks the thread as blocked and schedules another thread. A *V* operation works analogously; if the operation unblocks a thread that has higher priority than the invoker, the scheduler can block the invoking thread and run the newly-awakened one.

3.6 Device Drivers and Interrupt Handling

Each hardware device in the system has an associated semaphore used to communicate between the interrupt dispatcher component and the device driver component for the specific device.

In the portal table of each protection domain there are entries for the portals that corresponds to the machine's hardware interrupts. The Pebble nucleus includes a short trampoline function that handles all exceptions and interrupts. This code first determines the portal table of the current thread and then transfers control to the address that is taken from the corresponding entry in this portal table. The nucleus is oblivious to the specific semantics of the portal that is being invoked. The portal that handles the interrupt starts by saving the processor state on the invocation stack (see Section 5.1), then it switches to the interrupt stack and jumps to the interrupt dispatcher. In other words, this mechanism converts interrupts to portal calls.

The interrupt dispatcher determines which device generated the interrupt and performs a *V* operation on the device's semaphore. Typically, the device driver would have left a thread blocked on that semaphore. The *V* operation unblocks this thread, and if the now-runnable thread has higher priority than the currently running thread, it gains control of the CPU, and the interrupt is handled immediately. Typically, the priority of the interrupt handling threads corresponds to the hardware interrupt priority in order to support nested interrupts. The priority of the interrupt handling threads is higher than all other threads to ensure short handling latencies. In this way, Pebble unifies interrupt priority with thread priority, and handles both in the scheduler. A pictorial example of this process is found in Figure 1.

Note that Pebble invokes the interrupt dispatcher promptly for all interrupts, including low priority ones. However, the interrupt handling thread is scheduled only if its priority is higher than the currently running thread.

Only a small portion of Pebble runs with interrupts disabled, namely portal code, the interrupt dispatcher, and the scheduler. This is necessary to avoid race conditions due to nested exceptions.

3.7 Low and Consistent Interrupt Latency

Pebble provides low and consistent interrupt latency by design, since most servers (except the interrupt dispatcher and the scheduler) run with interrupts enabled. The interrupt-disabled execution path in Pebble is short, since portal code contain no loops, and the interrupt dispatcher and the scheduler are optimized for speed. User code cannot increase the length of the longest interrupt-disabled path, and thus cannot increase the interrupt latency. In previous work we included details on the interrupt handling mechanism in Pebble, along with measurements of the interrupt latency on machines with differering memory hierarchies [Bruno99]. In particular, the interrupt latency on the MIPS R5000 processor that is used in this paper is typically 1200-1300 cycles from the exception until the scheduling of the user-level handling thread.

3.8 Non-Stop Systems

Non-stop (or high-availability) systems are characterized by the ability to run continuously over extended periods of time and support dynamic updates. For example, some systems, such as telephone switches, are expected to run for years without unscheduled down time. Pebble is especially suited for these systems, since most system functionality may be replaced dynamically by loading new servers and modifying portal tables. The only component that cannot be replaced is the nucleus, which provides only minimal functionality.

4 Portals and Their Uses

Portals are used for multiple purposes in Pebble. In this section, we describe a few of their applications.

4.1 Interposition and Layering

One technique for building flexible system is to factor it into components with orthogonal functionality that can be composed in arbitrary ways. For example, distributed shared memory or persistent virtual memory can be implemented as a layer on top of a standard virtual memory service. Or, altered semantics can be offered by layering: the binary interface of one operating system can be emulated on another operating system by intercepting system calls made by an application written for the emulated system and implementing them through the use of native system calls.

The portal mechanism supports this development methodology very nicely. Because the portal mechanism is used uniformly throughout the system, and a portal performs a user-level to user-level transfer, service components can be designed to both accept and use the same set of portals.

For example, the primary task of a virtual memory manager is to accept requests for pages from its clients and service them by obtaining the pages from the backing store. When a client requests a page, the virtual memory manager would read the page from the backing store and return it to the client via a memory window operation. A standard virtual memory service implementation would support just this protocol, and would typically be configured with a user application as its client and the file system as its backing store server.

However, the backing store could be replaced with a distributed shared memory (DSM) server, which would have the same interface as the virtual memory manager: it would accept page requests from its client, obtain the pages from its backing store (although in this case the backing store for a page might be the local disk or another remote DSM server) and return the page to its client via a memory window operation. By implementing the DSM server using the standard virtual memory interface, it can be layered between the VM and the file system. Other services, such as persistent virtual memory and transactional memory, can be added this way as well.

When a page fault takes place, the faulting address is used to determine which portal to invoke. Typically a single VM fault handler is registered for the entire range of an application's heap, but this need not be the case. For example, a fault on a page in a shared memory region should be handled differently than a fault on a page in a private memory region. By assigning different portals to subranges of a protection domain's address space, different virtual memory semantics can be supported for each range.

4.2 Portals Can Encapsulate State

Because portal code is trusted, is specific to its portal, and can have private data, portal code can encapsulate state associated with the portal that need not be exposed to either endpoint. The state of the invoking thread is a trivial example of this: portal code saves the thread's registers on the invocation stack (see Section 5.1), and restores them when the thread returns. On the flip side, data used only by the invoked protection domain can be embedded in the portal where the invoker cannot view or manipulate it. Because the portal code cannot be modified by the invoking protection domain, the invoked protection domain is ensured that the values passed to it are valid. This technique frequently allows run-time demultiplexing and data validation code to be removed from the code path.

As an example, in Pebble, portals take the place of file descriptors. An `open()` call creates four portals in the invoking protection domain, one each for reading, writing, seeking and closing. The code for each portal has embedded in it a pointer to the control block for the file. To read the file, the client domain invokes the `read` portal; the portal code loads the control block pointer into a register and transfers control directly to the specific routine for reading the underlying object (disk file, socket, etc.). No file handle verification needs to be done, as the client is never given a file handle; nor does any demultiplexing or branching based on the type of the underlying object need to be done, as the appropriate read routine for the underlying object is invoked directly by the portal code. In this way, portals permit run-time checks to be "compiled out," shortening the code path.

To be more concrete, the `open()` call generates four consecutive portals in the caller's portal table. `Open()` returns a file descriptor, which corresponds to the index of the first of the four portals. The `read()`, `write()`, `seek()` and `close()` calls are implemented by library routines, which invoke the appropriate portals, as seen in Figure 2. `invoke_portal()` invokes the portal that is specified in its first argument. (Note that the portal code of `read` and `write` may map the buffer argument in a memory window to avoid data copying.)

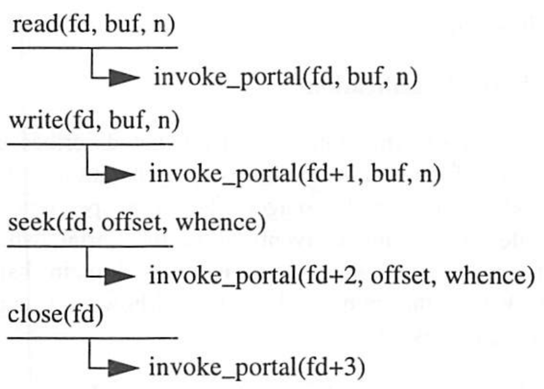

```
read(fd, buf, n)
        invoke_portal(fd, buf, n)
write(fd, buf, n)
        invoke_portal(fd+1, buf, n)
seek(fd, offset, whence)
        invoke_portal(fd+2, offset, whence)
close(fd)
        invoke_portal(fd+3)
```

Figure 2. Implementing file descriptors with portals

4.3 Short-Circuit Portals

In some cases the amount of work done by portal traversal to a server is so small that the portal code itself can implement the service. A short-circuit portal is one that does not actually transfer the invoking thread to a new protection domain, but instead performs the requested action inline, in the portal code. Examples include simple "system calls" to get the current thread's ID and read the high resolution cycle counter. The TLB miss handler (which is in software on the MIPS architecture, the current platform for Pebble) is also implemented as a short-circuit portal.

Currently, semaphore synchronization primitives are implemented by the scheduler and necessitate portal traversals even if the operation does not block. However, these primitives are good candidates for implementation as hybrid portals. When a *P* operation is done, if the semaphore's value is positive (and thus the invoking thread will not block), the only work done is to decrement the semaphore, and so there is no need for the thread to transfer to the scheduler. The portal code could decrement the semaphore directly, and then return. Only in the case where the semaphore's value is zero and the thread will block does the calling thread need to transfer to the scheduler. Similarly, a *V* operation on a semaphore with a non-negative value (*i.e.*, no threads are blocked waiting for the semaphore) could be performed in a handful of instructions in the portal code itself.

Although these optimizations are small ones (domain transfer takes only a few hundred cycles), operations that are on the critical path can benefit from even these small savings.

4.4 Portal Specification

The portal specification is a string that describes the behavior of the portal. It controls the generation of portal code by the portal manager. The portal specification includes the calling conventions of the portal, which registers are saved, whether the invoking domain shares a stack with the invoked domain, and how each arguments is processed.

The first character in the specification encodes the portal's stack manipulation. For example, `"s"` denotes that the invoking domain shares its stack with the invoked domain. `"n"` denotes that the invoked domain allocated a new stack. The second character specifies the amount of processor state that is saved or restored. For example, `"m"` denotes that only minimal state is saved, and that the invoking domain trusts the invoked domain to obey the C calling convention. `"p"` denotes that partial state is saved, and that the invoking domain does not trust the invoked domain to retain the values of the registers required by the C calling convention. The rest of the specification contains a sequence of single character function codes, that specify handling of the corresponding parameters. For example, the template `"smcwi"` specifies a shared stack, saving minimal state, passing a constant in the first parameter, passing a one-page memory window in the second parameter, and passing a word without transformation in the third parameter. This template is used by the `read` and `write` portals.

4.5 Portal Manipulations

As described earlier, portals are referred to by their index in the local portal table. A portal that is available in a particular portal table cannot be exported to other protection domains using this index. A protection domain may access only the portals in its portal table. These properties are the basis for Pebble safety. When a thread calls `fork()`, it creates a new thread that executes in the same protection domain as the parent. When a thread calls `domain_fork()`, it creates a new protection domain that has a copy of the parent domain's portal table. The parent may modify the child's portal table to allow portal interposition, which is described in Section 6.

5 Implementation Issues

In this section we discuss some of the more interesting implementation details of Pebble.

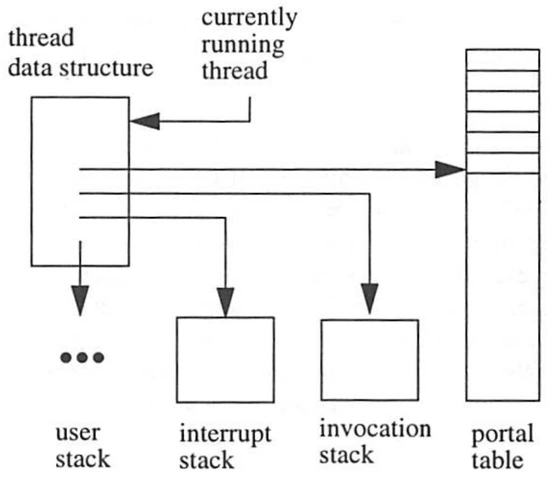


Figure 3. Pebble nucleus data structures

5.1 Nucleus Data Structures

The Pebble nucleus maintains only a handful of data structures, which are illustrated in Figure 3. Each thread is associated with a `Thread` data structure. It contains pointer to the thread's current portal table, user stack, interrupt stack and invocation stack. The user stack is the normal stack that is used by user mode code. The interrupt stack is used whenever an interrupt or exception occurs while the thread is executing. The interrupt portal switches to the interrupt stack, saves state on the invocation stack and calls the interrupt dispatcher server.

The invocation stack keeps track of portal traversals and processor state. The portal call code saves the invoking domain's state on this stack. It also saves the address of the corresponding return portal on the invocation stack. The portal return code restores the state from this stack.

The portal table pointer in the `Thread` data structure is portal table of the domain that the thread is currently executing in. It is changed by the portal call and restored by the portal return.

5.2 Virtual Memory and Cache

The virtual memory manager is responsible for maintaining the page tables, which are accessed by the TLB miss handler and by the memory window manipulation code in portals. The virtual memory manager is the only component that has access to the entire physical memory. The current implementation of Pebble does not support demand-paged virtual memory.

Pebble implementation takes advantage of the MIPS tagged memory architecture. Each protection domain is allocated a unique ASID (address space identifier), which avoids TLB and cache flushes during context switches. Portal calls and returns also load the mapping of the current stack into TLB entry 0 to avoid a certain TLB miss.

On the flip side, Pebble components run in separate protection domains in user mode, which necessitates careful memory allocation and cache flushes whenever a component must commit values to physical memory. For example, the portal manager must generate portal code so that it is placed in contiguous physical memory.

5.3 Memory Windows

The portal code that opens a memory window updates an access data structure that contains a vector of counters, one counter for each protection domain in the system. The vector is addressed by the ASID of the corresponding domain. The counter keeps track of the number of portal traversals into the corresponding domain that passed this page in a memory window. This counter is incremented by one for each portal call, and is decremented by one for each portal return. The page is accessible if the counter that corresponds with the domain is greater than zero. We must use counters and not bit values for maintaining page access rights, since the same page may be handed to the same domain by multiple concurrent threads.

The page table contains a pointer to the corresponding access data structure, if any. Only shared pages have a dedicated access data structure.

The portal code does not load the TLB with the mapping of the memory window page. Rather, the TLB miss handler consults this counter vector in order to verify the access rights to this page. This arrangement saves time if the shared window is passed to another domain without being touched by the current domain. The portal return code must remove the corresponding TLB entry when the counter reaches zero.

5.4 Stack Manipulations

The portal call may implement stack sharing, which does not require any stack manipulations. The invoked domain just uses the current thread's stack.

If the portal call requires a new stack, it obtains one from the invoked domain's stack queue. In this case, the invoked protection domain must pre-allocate one or more stacks and notify the portal manger to place them in the domain's stack queue. The portal call dequeues a new stack from the invoked domain's stack queue. If the stacks queue is empty, the portal calls the scheduler and waits until a stack becomes available. The portal return enqueues the released stack back in the stack queue. If there are any threads waiting for the stack, the portal return calls the scheduler to pick the first waiting thread and allow it to proceed in its portal code.

The portal that calls the interrupt dispatcher after an interrupt switches the stack to the interrupt stack, which is always available in every thread.

5.5 Footprint

The Pebble nucleus and the essential components (interrupt dispatcher, scheduler, portal manager, real-time clock, console driver and the idle task) can fit into about 70 pages (8KB each). Pebble does not support shared libraries yet, which cause code duplication among components. Each user thread has three stacks (user, interrupt and invocation) which require three pages, although the interrupt and invocation stacks could be placed on the same page to reduce memory consumption. In addition, fixed size pages inherently waste memory. This could be alleviated on segmented architectures.

6 Portal Interposition

An important aspect of component-based system is the ability to *interpose* code between any client and its servers. The interposed code can modify the operation of the server, enforce safety policies, enable logging and error recovery services, or even implement protocol stacks and other layered system services.

Pebble implements low-overhead interposition by modifying the portal table of the controlled domain. Since all interactions between the domain and its surroundings are implemented by portal traversals, it is possible to place the controlled domain in a comprehensive sandbox by replacing the domain's portal table. All of the original portals are replaced with portal stubs, which transfer to the interposed controlling domain. The controlling domain intercepts each portal traversal that takes place, performs whatever actions it deems necessary, and then calls the original portal. Portal stubs pass their parameters in the same way as the original portals, which is necessary to maintain the semantics of the parameter passing (*e.g.* windows). Actually, portal stubs are regular portals that pass the corresponding portal index in their first argument. The controlling domain does not have to be aware of the particular semantics of the intercepted portals; it can implement a transparent sandbox by passing portal parameters verbatim.

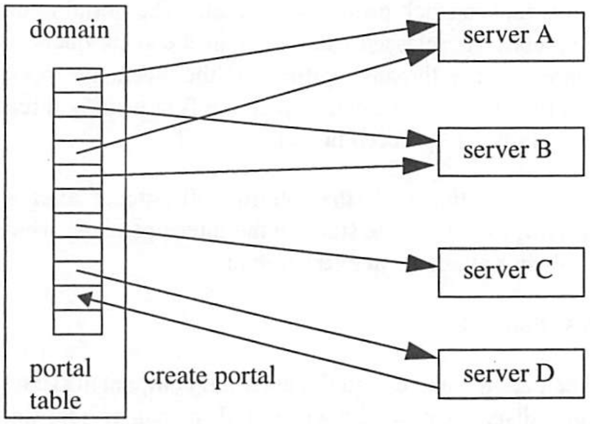
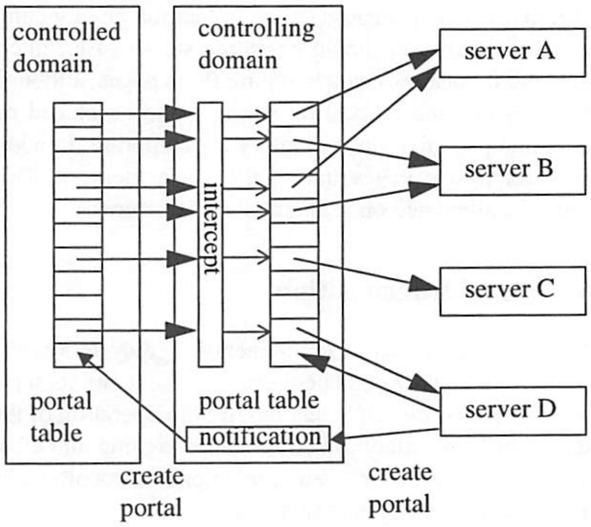


Figure 4. Original portal configuration (above) and with portal interposition (below)



The top diagram of Figure 4 illustrates the configuration of the original portal table without interposition, where the domain calls its servers directly. The bottom diagram shows the operation of portal interposition. In this case, all of the portals in the controlled domain call the controlling domain, which makes the calls to the servers.

However, one-time modification of the controlled domain's portal table is not enough. Many servers create new portals dynamically in their client's portal table, and then return an index to the newly created portal back to the client. Since the controlling domain calls the server, the server creates new portals in the controlling domain's table. The controlling domain is notified by the portal manager that a new portal was created in its portal table. The notification portal completes the process by creating a portal stub in the controlled domain's table with the same index as in controlling domain table. The portal stub calls the controlling domain and passes the parameters in the same way as the original portal. In this way, the controlling domain implements a robust sandbox around the controlled domain, without actually understanding the semantics of the controlled domain portals.

There are a few comments about this interposition mechanism. First, the controlled domain cannot detect that its portals are diverted nor can it thwart the interposition in any way. This mechanism is similar to the Unix I/O redirection, in which a child process accesses standard file descriptor (*e.g.*, 0, 1 and 2), which are redirected by the parent process. Second, portal interposition is more comprehensive than Unix I/O redirection, since we can control *all* interactions between the controlled domain and its environment. Third, interposition can be recursive: a controlling domain interposes the portals of a child domain, which does the same to its child, *ad infinitum*. The last comment deals with the semantics of certain system services, like `fork()` and `sbrk()`, which change the internal state of the calling domain; these are somewhat tricky to implement in the face of transparent interposition. We have had to make special accommodations to allow the controlling domain to issue them on behalf of the controlled domain.

6.1 Implementing a Transparent Sandbox by Portal Interposition

The Appendix contains a code excerpt from a program that implements a transparent sandbox around its child domain. The program counts the number of times each portal was called by the child domain, and completes all child portal traversals by calling the appropriate server. It is a fully functional program; we omitted only error handling code, due to space constraints. When run on our test hardware (see Section 7, below) the overhead of this process is 1511 machine cycles for one iteration (two `sem_wait()` and two `sem_post()`), which is roughly twice the execution time of the original code without interposition.

The program starts by calling `portal_notify()`, which registers the routine `notify()` with the portal manager. Any modification to the calling domain's portal table will call `notify()` immediately even before the portal that caused it has returned. `Portal_notify()` is necessary to handle any portal call that the parent executed on behalf of the child which created a new portal in the parent's portal table. This new portal should be replicated also in the child's portal table to ensure correct operation. The above situation occurs in the example when the parent executes `sem_create()` on behalf of the child.

The `notify()` routine receives the template of the newly created portal and its position in the portal table. It creates a portal in the child's portal table at the same position. The portal's template is modified to pass the portal number as the first argument.

The program proceeds to create a child domain by `domain_fork()`. The child starts with a copy of the parent's portal table. However, all of the entries in the child's portal table now point at the `intercept()` routine in the parent domain. The first argument to the `intercept()` routine is the index of the called portal in the portal table. This routine increments the counters and then performs the required action by invoking the portal with the same index in the parent domain. `invoke_portal()` let applications invoke a specific portal in the caller's portal table. The `intercept()` routine assumes that portals have no more than five parameters.

The child domain executes the `measure()` routine, which measures the execution time of a semaphore ping-pong between two threads in the same domain. The `hrtime()` function returns the current value of the high-resolution timer, which is incremented every two machine cycles. `Measure()` creates two semaphores by calling `sem_create()`. The scheduler creates two new portals for each semaphore in the parent domain, which calls `notify()` to create the corresponding stubs in the child domain's portal table.

7 Performance Measurements

In this section we measure the performance of Pebble and, where possible, compare it with OpenBSD running on the same hardware. The test hardware is an Algorithmics P-5064 board, which includes a 166 MHz MIPS R5000 processor with 32 KB instruction + 32 KB data level one cache (two way set associative), one megabyte integrated level two cache and 64MB of memory. We ran version 2.4 of OpenBSD.

Times were measured using the high-resolution on-chip timer, which is incremented every two clock cycles. All results are presented in terms of elapsed machine cycles, not elapsed time, as our tests generally fit into the level one or level two cache. As long as cache memory speed scales with processor speed, cycle-based results will remain meaningful. To convert cycle counts to elapsed time, multiply by the cycle time (6 ns).

As the code size of Pebble is very small, and the cache associativity of the level one cache is low (two-way), the performance of Pebble is very dependent on how code and data is placed in the cache. Out of a sense of fairness, in our experiments we specifically do not make any attempt to control cache layout. We believe that with careful tuning of the cache layout, we could reduce the number of cache misses and conflicts. Given the performance results we have seen to date, we have felt little need to go to this effort.

The context switch, pipe latency, and semaphore latency tests were adapted from the hBench:OS test suite [Brown98]. All tests on Pebble were run 10,000 times. The context switch and pipe latency times presented for OpenBSD were the 80% trimmed mean (excluding the smallest 10% and largest 10% of the measurements) of twenty results of 10,000 iterations, as per the hBench:OS measurement methodology. In all cases the standard deviation for Pebble measurements was less than 1%.

7.1 IPC

A naive implementation of inter-process communication (IPC) will emulate the behavior of a remote procedure call (RPC), marshalling all arguments into a buffer, copying the buffer from the invoking protection domain to the invoked protection domain, unmarshalling them, and then calling the server function. Several common optimizations can be performed that greatly improve the performance of IPC.

First, the amount of data transmitted in an IPC follows a bimodal distribution [Bershad89]; either a small number of bytes are sent (in which case they can be passed in registers) or a large number of bytes are sent (in which case it may make more sense to transfer the data using virtual memory mapping operations).

In this test we measure the cost of performing an IPC when all data fits into registers, when a one-page memory window is passed to the invoked domain (but the invoked domain does not access the page), and when the one-page memory window is written by the invoked domain. Because virtual memory and the TLB are managed in software on the MIPS, the memory management unit is not involved if when passing a memory window if the window is never used, although there is some additional portal overhead. When the window is used in the invoked domain, a TLB fault takes place, and the memory management unit comes into play. Moreover, the portal code may have to remove the resulting TLB entry on return.

Simply measuring the per-leg cost of an IPC between two domains does not tell the entire story. In a system that has been factored into components, we may find that a client request to service *A* causes *A* to make a request of *A'*, *A'* to make a request of *A''*, and so on, until the initial request is finally satisfied. For example, a client page fault generates a request to its VM service, then makes a request of the file system, which then makes a request the disk driver to bring the page into memory. Although simple IPC between two protection domains must be cheap, it is also critical that when a cascade of IPCs takes place performance does not drop precipitously.

In this test we measure the time to perform an IPC to the same domain and return (*A*→*A*→*A*), the time required to perform an IPC to a second domain and return (*A*→*B*→*A*), an IPC involving three domains (*A*→*B*→*C*→*B*→*A*) and so on, up to a total of eight domains. We used the portal specification `"npciii"` (no window) and `"npcwii"` (with memory window), which means that a new stack was allocated on call and reclaimed on the return. Also, all processor registers that should be preserved across calls according to the C calling convention were saved on call and restored on return. See Section 4.4 for a description of portal specification. The results are presented as the per-leg (one-way) time, in cycles.

As a point of comparison, we included the time required to perform a "null" short-circuit portal traversal (user level → nucleus → user level). This is the Pebble equivalent to a "null" system call, and can be thought of as the minimum time required to enter and leave the nucleus. Results of these tests are found in Table 1. In all cases.

n domains	no window	window	window + fault
short-circuit	45	—	—
1	114	133	135
2	114	134	185
4	118	139	190
8	133	153	209

Table 1. IPC in Pebble, new stack and partial save, All times in CPU cycles, the mean of 10,000 runs.

parameters are passed only in registers and not on the stack.

We see that the times per leg with no window and with an unused window remains roughly constant as the number of domains traversed increases, at about 114 and 135 cycles; the overhead of passing a window through a portal is thus 21 machine cycles. The time per leg increases above 4 domains due to cache contention. When the memory window is used, the cost increases by about 50 cycles, which is the time required to handle a TLB fault and then remove the TLB entry on return from the IPC. The one outlier is in the single domain case, where there is no TLB fault at all; this is because the page is already mapped in the domain (as there is only one domain).

An optimization can be performed if the invoking domain trusts the invoked domain (as would be the case with an application invoking a system service). The two can share a stack, saving the costs of allocating a stack from a pool in the invoked protection domain and copying data to the new stack. Also, no additional processor registered are saved on the call, since the invoking domain trusts the invoked domain to save and restore those registers. We used the portal specifications `"smciii"` and `"smcwii"`. Even in the tested case, where no data is passed on the stack, this optimization has a significant performance benefit, as seen in Table 2.

n domains	no window	window	window + fault
1	95	115	118
2	95	116	168
4	95	116	168
8	98	120	182

Table 2. IPC in Pebble, shared stack and minimal save. In CPU cycles, the mean of 10,000 runs

The savings of this optimization are measured here to be about 20 cycles, which reduces the per-leg time by 17%. In addition, by sharing stacks between invoking and invoked protection domains, the number of stacks, and hence amount of memory, needed by the system is decreased, which is an absolute good.

Pebble IPC time is slightly higher than Aegis, an exokernel, on MIPS processors [Engler95]. Aegis performs a minimal one-way protected control transfer in about 36 cycles on MIPS R2000 and R3000 processors, and performs a null system call without a stack in about 40 cycles. Pebble's IPC takes longer since it maintains an invocation stack, which enables easy scheduling of the thread.

7.2 Context Switch

As described above, portal traversal does not involve a scheduling decision. In this section we show the cost of a context switch in Pebble.

We measure Pebble context switch cost in two ways, first using Pebble's explicit yield primitive, and then by passing a one-byte token around a ring of pipes. The latter test was derived from hBench:OS, and was used to compare the performance of Pebble with OpenBSD. In both cases a number of protection domains, with a single thread each, are arranged in a ring, and scheduled in turn. Measurements are found in Table 3.

n domains	Pebble yield	Pebble pipe	OpenBSD pipe
2	425	411	1195
4	549	963	2093
8	814	1162	2179

Table 3. Context switch times, Pebble vs. OpenBSD. In CPU cycles, the mean of at least 10,000 runs.

We see that the cost of an explicit yield increases with the number of protection domains, up to a certain point, and then levels off. As the work done by the scheduler in this case is independent of the number of processes (it simply selects the next thread from the ready queue), the increase in time is due to cache effects: as we grow out of the level one cache, we rely more on the level two cache, to the point where we are running almost entirely out of the level two cache (at six protection domains). We would expect to see a similar jump at the point where we begin to overflow the one-megabyte level two cache.

The OpenBSD pipe test shows similar behavior, leveling off at four protection domains and roughly 2200 machine cycles.

7.3 Pipe Latency

This test measures the time required to pass a single byte through pipes connecting a ring of processes. Each value represents the time to transfer one byte between two adjacent processes, and includes the context switch time. By measuring the time required to transmit a single byte, we capture the overhead associated with using pipes; the more data that is sent, the more the data copy time will mask pipe costs. Results are found in Table 4.

n domains	Pebble pipe	OpenBSD pipe
2	1310	3088
4	1914	3979
8	2061	4055

Table 4. Pipe latency, Pebble vs. OpenBSD. In CPU cycles, the mean of at least 10,000 runs.

We see that, as with the context switch times, the OpenBSD pipe time increases up to five domains, and then levels off. The difference between the numbers in Table 4 and Table 3 gives us the time required to transfer data through a pipe on each system. On OpenBSD the pipe overhead is roughly 2000 cycles; on Pebble it is approximately half that.

7.4 Semaphore Acquire/Release

This test is very similar to the test in Section 7.3, but instead of using pipes we use semaphores. A number of processes are arranged in a ring, and are synchronized by means of n semaphores. Each process performs a *V* operation on its right semaphore and then a *P* operation on its left semaphore. Each value in the table represents the time to release a semaphore in process *p* and acquire it in process *(p + 1) mod n* around a ring of n processes, including the context switch time. Results are found in Table 5.

n domains	Pebble semaphore	OpenBSD semaphore
2	781	2275
4	942	3415
8	1198	5091

Table 5. Semaphore acquire/release, Pebble vs. OpenBSD. In CPU cycles, the mean of 10,000 runs.

When there are two processes the difference between Pebble and OpenBSD is roughly 1500 cycles, 1000 cycles of which can be attributed to the difference in context switch times. As the number of domains (and thus semaphores) increases, the difference widens; because Pebble's semaphores are a highly optimized key system primitive, and OpenBSD's semaphores are not, we believe that this is due to a restriction in the implementation of OpenBSD semaphores, and is not a reflection of the difference in system structure.

7.5 Portal Generation

Table 6 shows the portal generation time for two typical portals. This is the time measured by an application program, including all overheads incurred by the portal manager. The first portal (with specification "`smcii`") is typically used to call a trusted server with only integer arguments. The second portal (with specification "`npcwi`") is typically used to call an untrusted server with a memory window argument. See Section 4.4 for additional explanations of portal specifications.

portal spec.	portal len (instr.)	time (cycles)	cycles per instr.
smcii	64	7282	114
npcwi	112	8593	77

Table 6. Portal generation time.

Table 6 indicates that portal generation time is relatively fast. An examination of the portal manager reveals that portal generation time includes a large fixed overhead for interpretation of the specification string and for cache flushing. We can reduce this time by employing various techniques used for run-time code generation, *e.g.*, the techniques used by VCODE [Engler96].

8 Status and Future Work

The Pebble nucleus and a small set of servers (scheduler, portal manager, interrupt dispatcher, and minimal VM) and devices (console and clock) currently run on MIPS-based single-board computers from Algorithmics. We support both the P-4032 (with QED RM5230 processor) and P-5064 (with IDT R5000 or QED RM7000 processors). We are currently porting Ethernet and SCSI device drivers to Pebble.

Next we plan to port Pebble to the Intel x86 to verify that Pebble mechanisms and performance advantages are indeed architecture independent. We also plan to implement a demand-paged virtual memory system. Building a high-performance VM system for Pebble is a challenge, since the servers cannot (and should not) share data structures freely. We also plan to port a TCP/IP stack to Pebble and compare its performance with similar user-level protocol stacks.

In addition to the Intel x86 port, we plan to port to a symmetric multiprocessor and to an embedded processor such as the StrongARM. We also plan to investigate the various processor architecture support for component-based systems such as Pebble.

9 Summary

Pebble provides a new engineering trade-off for the construction of efficient component-based systems, using hardware memory management to enforce protection domain boundaries, and reducing the cross domain transfer time by synthesizing custom portal code. Pebble enhances flexibility by maintaining a private portal table for each domain. This table can be used to provide different implementations of system services, servers and portal interposition for each domain. In addition, portal interposition allows running untrusted code in a robust sandbox with an acceptable overhead while using unsafe languages such as C.

Having a small nucleus with minimal functionality enhances system modularity, while it enables non-stop systems to modify their behavior by integrating new servers on-the-fly.

In this paper we showed that Pebble is much faster than OpenBSD for a limited set of system-related micro-benchmarks. Pebble efficiency does not stem from clever low-level highly-optimized code; rather it is a natural consequence of custom portal synthesis, judicious processor state manipulations at portal traversals, encapsulating state in portal code, and direct transfer of control from clients to their servers without scheduler intervention.

Pebble can be used to build systems that are more flexible, as safe as, and have higher performance than conventionally constructed systems.

Acknowledgments

The authors would like to thank the anonymous referees for their insightful comments.

References

[Accetta86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proc. Summer 1986 USENIX Conf.*, pp. 93–112 (1986).

[Bershad89] B. Bershad, T. Anderson, E. Lazowska, H. Levy, "Lightweight Remote Procedure Call," *Proc. 12th SOSP*, pp. 102–113 (1989).

[Bershad95] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, S. Eggers, "Extensibility, Safety, and Performance in the SPIN Operating System," *Proc. 15th SOSP*, pp. 267–284 (1995).

[Brown98] A. Brown, M. Seltzer, "Operating System Benchmarking in the Wake of lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture," *Proc. 1997 SIGMETRICS*, pp. 214–224 (1997).

[Bruno99] J. Bruno, J. Brustoloni, E. Gabber, A. Silberschatz, C. Small, "Pebble: A Component-Based Operating System for Embedded Applications," *Proc. USENIX Workshop on Embedded Systems,* Cambridge, MA (1999).

[Custer92] H. Custer, *Inside Windows NT,* Microsoft Press, Redmond, WA (1992).

[Dijkstra68] E. W. Dijkstra, "The Structure of "THE" Multiprogramming System," CACM, Volume 11, Number 5, pp. 341-346 (1968).

[Dorward97] S. Dorward, R. Pike, D. Presotto, D. Ritchie, H. Trickey, P. Winterbottom, "Inferno," *Proc. IEEE Compcon 97,* pp. 241–244 (1997).

[Engler95] D. Engler, M. Frans Kaashoek, J. O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management", *Proc. 15th SOSP,* pp. 251-266 (1995).

[Engler96] D. Engler, "VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System", *Proc. Conference on Programming Language Design and Implementation (PLDI'96)*, pp. 160-170 (1996).

[Ford96] B. Ford, M. Hibler, J,. Lepreau, P. Tullmann, G. Back, S. Clawson, "Microkernels Meet Recursive Virtual Machines," *Proc. 2nd OSDI*, pp. 137–151 (1996).

[Golub90] D. Golub, R. Dean, A. Forin, R. Rashid, "UNIX as an Application Program," *Proc. 1990 Summer USENIX*, pp. 87–96 (1990).

[Gosling96] J. Gosling, B. Joy, G. Steele, *The Java™ Language Specification*, Addison-Wesley, Reading, MA (1996).

[Helander98] J. Helander and A. Forin, "MMLite: A Highly Componentized System Architecture", *Proc. 8th ACM SIGOPS European Workshop,* Sintra, Portugal (1998).

[Intel94] Intel Corp., *Pentium Family User's Manual Volume 3: Architecture and Programming Manual* (1994).

[Kaashoek97] M. F. Kaashoek, D. Engler, G. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, "Application Performance and Flexibility on Exokernel Systems," *Proc. 16th SOSP,* pp. 52–65 (1997).

[Liedtke95] J. Liedtke, "On Micro-Kernel Construction," *Proc. 15th SOSP,* pp. 237–250 (1995).

[Liedtke97] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, T. Jager, "Achieved IPC Performance," *Proc. 6th HotOS*, pp. 28–3 (1997).

[Massalin92] H. Massalin, *Synthesis: An Efficient Implementation of Fundamental Operating System Services*, Ph.D. thesis, Columbia University Department of Computer Science, New York, NY (1992).

[Mitchell94] J. G. Mitchel *et al*, "An Overview of the Spring System", *Proc. Compcon Spring 1994*, pp. 122-131 (1994).

[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, "Plan 9 from Bell Labs," *Proc. Summer 1990 UKUUG Conf.*, pp. 1–9 (1990).

[Probert91] D. Probert, J. Bruno, M. Karaorman, "SPACE: A New Approach to Operating System Abstractions," *Proc. Intl. Workshop on Object Orientation in Operating Systems (IWOOS),* pp. 133–137 (1991), Also available on-line at ftp.cs.ucsb.edu/pub/papers/space/iwooos91.ps.gz

[Pu95] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, K. Zhang, "Optimistic Incremental Specialization: Streamlining a Commercial Operating System," *Proc. 15th SOSP,* pp. 314–324, (1995).

[Rozier88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, W. Neuhauser. "Chorus Distributed Operating System." *Computing Systems 1*(4), pp. 305–370 (1988).

[Seltzer96] M. Seltzer, Y. Endo, C. Small, K. Smith, "Dealing With Disaster: Surviving Misbehaved Kernel Extensions," *Proc. 2nd OSDI*, pp. 213–227 (1996).

[Veitch96] A. C. Veitch and N. C. Hutchinson, "Kea - A Dynamically Extensible and Configurable Operating System Kernel", Proc. 3rd Conference on Configurable and Distributed Systems (ICCDS'96), Annapolis, Mariland (1996).

Appendix: Implementing a Transparent Sandbox by Portal Interposition

```
/* see Section 6.1 for explanations */
#include <pebble.h>

#define  N  10000

int child_asid;
int count[NPORTALS];

/* child domain runs this routine */
void measure(void)
{
   int code, i;
   unsigned long long start, elapsed;
   int sem_id1, sem_id2;

   /* create semaphores */
   sem_id1 = sem_create(0);
   sem_id2 = sem_create(0);

   /* create child thread in the same
      domain */
   if ((code = fork()) == 0) {
      /* child thread wakes parent */
      sem_post(sem_id2);

      for (i = 0;; i++) {
         sem_wait(sem_id1);
         sem_post(sem_id2);
      }

      /* never reached */
      exit(1);
   }

   /* parent thread waits until child
      is active for accurate timing */
   sem_wait(sem_id2);

   /* time semaphore ping-pong with
      child */
   start = hrtime();

   for (i = 0; i < N; i++) {
      sem_post(sem_id1);
      sem_wait(sem_id2);
   }
   elapsed = 2*(hrtime() - start);

   printf("each iteration: %d \
      cycles\n", (int)(elapsed/N));
}

void dump_counters(void)
{
   int i;

   for (i = 1; i < NPORTALS; i++)
      if (count[i] != 0)
         printf("portal %d called %d\
            times\n", i, count[i]);
}

/* parent domain intercepts child por-
tal call */
int intercept(int id, int p1, int p2,
         int p3, int p4, int p5)
{
   count[id]++;
   if (id == SYS_EXIT)
      dump_counters();

   return invoke_portal(id, p1, p2,
                        p3, p4, p5);
}

/* parent domain gets notification */
int notify(int asid, int id,
      char *template)
{
   char s[NAMELEN];

   sprintf(s, "sm=%s", template+3);
   portal_create_child(child_asid,
      id, s, 0, intercept);
   return 0;
}

void main(void)
{
   portal_notify(notify);

   child_asid =
      domain_fork(intercept);
   if (child_asid == 0) {
      /* child domain */
      measure();
      exit(0);
   }

   /* parent waits here until child
      exits */
   sem_wait(sem_create(0));

   exit(0);
}
```

Linking Programs in a Single Address Space

Luke Deller, Gernot Heiser
School of Computer Science & Engineering
University of New South Wales
Sydney 2052, Australia
{luked,gernot}@cse.unsw.edu.au, http://www.cse.unsw.edu.au/~disy

Abstract

Linking and loading are the final steps in preparing a program for execution. This paper assesses issues concerning dynamic and static linking in traditional as well as single-address-space operating systems (SASOS). Related loading issues are also addressed. We present the dynamic linking model implemented in the Mungi SASOS and discuss its strengths and limitations. Benchmarking shows that dynamic linking in a SASOS carries significantly less overhead than dynamic linking in SGI's Irix operating system. The same performance advantages could be achieved in Unix systems, if they reserved a portion of the address space for dynamically linked libraries, and ensured that each library is always mapped at the same address.

1 Introduction

Single-address-space operating systems (SASOS) make use of the wide address spaces offered by modern microprocessor architectures to greatly simplify sharing of data between processes [WSO+92, CLBHL92, RSE+92, CLFL94]. This is done by allocating *all data* in the system, whether transient or persistent, at a unique and immutable virtual address. As a result, all data is *visible* to every process, and no pointer translations are necessary for sharing arbitrary data structures. While the global address space makes all data addressable, a protection system ensures that *access* only succeeds when authorised. Protection relies on the fact that a process can only access a page if it has been mapped to a RAM frame, by the operating system loading an appropriate entry into the translation lookaside buffer (TLB). The operating system thus has full control over which parts of the single address space are accessible to a given process.

As persistent data, i.e., data whose lifetime is independent of that of the creating process, is always mapped into the virtual address space (at an immutable address), SASOS do not need a file system. For compatibility with other systems, a file system interface can be provided, of course, but it represents nothing more than a different way to access virtual memory. All disk I/O is done by the virtual memory paging system.

Such a SASOS is rather different from a system like MacOS, which also shares an address space between all executing programs. The main difference (other than the absence of memory protection) is that the latter system does not ensure that a data item has a unique address for its lifetime. For example, files can be mapped into memory, but each time a file is opened, it will be mapped at a different address. As each object in a SASOS has an immutable address, pointers are perfect object references which do not lose their meaning when passed between processes or stored in files. This greatly facilitates sharing of data between programs in a SASOS, as any data item can always be uniquely identified by its virtual memory address.

Besides sharing, the single address space also significantly simplifies system implementation [WM96] and improves performance even for applications which are not aware of the single address space [HEV+98]. However, the changed notion of address spaces makes it necessary to rethink a number of issues relating to how the system is used in practice. These include preparing programs for execution: how to bind together separately compiled program components ("linking"), and how to get an executable program into a state where the CPU can execute its instructions ("loading").

In this paper we examine the issues of linking and loading of programs in traditional Unix systems, as well as in SASOS. We present the model of (dynamic) linking used in the Mungi SASOS [HEV+98] which is under development at UNSW. The implementation of dynamic linking in Mungi is discussed and its performance compared to that of Unix operating systems.

2 Linking in traditional operating systems

Traditional operating systems, such as Unix systems, generally feature two forms of linkage, static and dynamic.

During *static linking*, all code modules are copied into a single executable file. The location of the various modules within that file implies their location in the memory image (and hence in the address space) during execution, and therefore target addresses for all cross-module references can be determined and inserted into the executable file by the linker.

A *dynamic linker*, in contrast, inserts symbolic references to (library) modules in the executable file, and leaves these to be resolved at run time. In operating systems conforming to the Unix System-V interface [X/O90], such as SGI's Irix, DEC's Digital Unix or Sun's Solaris-2, this works as follows.

For each library module to be linked, the linker allocates a *global offset table* (GOT).[1] The GOT contains the addresses of all dynamically linked external symbols (functions and variables) referenced by the module.[2] When a program referencing such a dynamically linked module is loaded into memory for execution, the imported module is loaded (unless already resident) and a region in the new process' address space is allocated in which to map the module. The loader then initialises the module's GOT (which may require first loading other library modules referenced by the module just loaded).

A variant of this is *lazy loading*, where library modules are not actually loaded until accessed by the process. If lazy loading is used, a module's GOT is initialised at module load time with pointers to stub code. These stubs, when called, invoke the dynamic loader, which loads the referenced module and then replaces the respective entries in the GOT by the addresses of the actual variables and functions within the newly loaded module.

Dynamic linking has a number of advantages over static linking:

1. Library code is not duplicated in every executable image referencing it. This saves significant amounts of disk space (by reducing the size of executable files) and physical memory (by sharing library code between all invocations). These savings can be very substantial and have significant impact on the performance of the virtual memory system.
2. New (and presumably improved) versions of libraries can be installed and are immediately usable by client programs without requiring explicit relinking.
3. Library code which is already resident can be linked immediately, thus reducing process startup latency.

Lazy loading further reduces startup cost, at the expense of briefly stalling execution when a previously unaccessed library module requires loading. For libraries which are only occasionally used by the program, this results in an overall speedup for runs which do not access the library. However, this comes at a cost: Should the referenced library be unavailable (e.g., by having been removed since program link time) this may only become evident well into the execution of the program. Many users will prefer finding out about such error conditions at program startup time.

The main drawbacks of dynamic linking are:

1. Extra work needs to be done at process instantiation to set up the GOT. However, this overhead is easily amortised by loading much less code in average, as some libraries will already be resident.
2. If a dynamic library is (re)moved between link and load time, execution will fail. This is the main reason that Unix systems keep static linking as an option.
3. The location of a dynamically linked module in the process' address space is not known until run time, and the same module will, in general, reside at different locations in different clients' address spaces. This requires that dynamically linked libraries only contain *position-independent code*.[3] Position independence requires that all jumps must

[1] The "global offset table" is Irix terminology, Digital Unix calls it *global address table*.

[2] A further level of indirection is used when an entry references a module exporting a large number of symbols, and for efficiency reasons local symbols are also included.

[3] This is different from *relocatable code* normally produced by compilers. Relocatable code contains addresses which are relative to some yet unresolved symbols. The linker resolves these and replaces them by absolute addresses.

be PC-relative, relative to an index-register containing the base address of the module, or indirect (via the GOT).

The main cost associated with position-independent code is that of locating the GOT. *Every* exported ("public") function in the module must first locate the module's GOT. The GOT is allocated at a constant offset from the function's entry point (determined by the linker) so the function can access it using PC-relative addressing. This code must be executed at the beginning of every exported function. In addition, there is an overhead (of one cycle) for calling the function, as an indirect jump must be used, rather than jumping to an absolute address. These costs will be examined further in Section 6.

Note that the GOT is an example of *private static data*, i.e., data belonging to a module but not shared between different instantiations of the module. Any variable declared "extern" or "static" in the library and not used read-only falls into that category.

It is interesting to note that Digital Unix' *quickstart* facility [DEC94] tries to avoid some of the problems of dynamic linking by reserving a system-wide unique virtual address range for dynamically linked libraries in the Alpha's large address space. This reduces process startup costs by the ability to easily share an already loaded library without address conflicts. However, address clashes cannot be completely avoided, as there is nothing to *enforce* unique virtual address for every library — only a SASOS can give such a guarantee. Consequently, Digital Unix still needs to use position-independent code and pay the overhead this implies. However, Digital's attempt to simulate a single address space for libraries is a good indication of some of the advantages a SASOS offers.

3 Linking in single-address-space systems

A single address space simplifies many things and complicates a few; linking is no exception. Generally speaking, the single address space makes it easy to share data, and difficult not to share. The latter implies some special effort to avoid sharing of private static data.

3.1 Static linking in a SASOS

Static linking by copying all libraries into a single executable is possible in a SASOS exactly as in Unix systems. Consequently, standard static linking in Mungi has the same drawbacks as in Unix: excessive disk and memory use, as well as the requirement to re-link in order to utilise new library versions. Therefore, alternative linking schemes are desirable.

Owing to the fact that all objects in the single address space are at any time fully and uniquely identified by their unchanging address, copying library modules is unnecessary when creating an executable program. Instead, libraries can be executed *in-place*, and the linker only needs to replace references to library modules with (absolute) addresses. No position-independent code is needed, avoiding that source of efficiency loss. This scheme, called *global static linking* was proposed by Chase [Cha95] for the Opal SASOS.

Global static linking is fast and has some of the attractive features of dynamic linking in Unix systems, in particular automatic code-sharing. However, the scheme has two significant drawbacks which limit its applicability in practice:

- As it is a form of static linking, new versions of libraries cannot be used unless programs are re-linked. Note that it is not possible to update a library in-place, as this would break entrypoint addresses in all programs which linked that library. While this could be circumvented by accessing all entrypoints via a jump table, that would not help with currently executing programs which use the library — they would have to terminate prior to replacing the library. To maintain smooth operation, a new library version must be created at a different virtual address, with a naming service pointing to the latest version to be used by the linker.
- Global static linking does not allow private static data. Such data needs to reside at an address where it can be found by the library code, but that address must be different for different instantiations of the library (or the data would not be private to the invoking process).

Private static data must reside in a separate data segment, which must be set up separately for each client process. Similar to dynamic linking in Unix, the problem is how to tell the library code where to find the data segment.

Chase suggests a variation of the GOT used by Unix dynamic linkers: Each process allocates a table for each module containing the addresses of that module's private static data. A dedicated register, the *global pointer*, is loaded with the base address of the address table of the presently executing library module. The difficulty is in loading the global pointer with the correct address. Unlike the Unix case, this cannot be done by PC-relative addressing, as the offset differs between instantiations.

Chase favours (but apparently did not implement) an approach where the called function looks up the correct global-pointer value in a process-specific table (accessed via its thread descriptor, which in Opal is reachable from the stack pointer). The table is indexed by a slot number which is statically assigned to the module containing the function. Given that it is impractical to make this (per-process) table very big, this imposes serious limitations on the use of modules containing private static data — each process can only use a small number of such libraries and even then, clashes between the statically assigned slot numbers preclude importing certain combinations of library modules. Furthermore, at least two memory reads plus some arithmetic is required to obtain the global pointer value on each call across module boundaries.

3.2 Dynamic linking in a SASOS

Even if the problem with private static data is resolved (or ignored), any form of static linking retains one major drawback: A new version of a library cannot be incorporated without relinking client programs. This can only be achieved by dynamic linking.

The IBM AS/400, a SASOS which, in its former guise as System/38 [Ber80], goes back to the mid 1970's, originally *only* supported dynamic linking ("late binding" in IBM terminology) [Sol96]. Static linking ("early binding by copy") was only introduced in 1993 *for performance reasons* resulting from the calling overhead. At the same time they introduced "early binding by reference" which essentially is global static linking. According to Soltis, while there is some initialisation overhead when first accessing a bound-by-reference library module, performance of subsequent calls are "about the same" as in the bound-by-copy case. No further information could be found on how the AS/400 implements dynamic linking, but the reference to performance problems of dynamic linking seems to indicate that it does not have a particularly good solution.

Roscoe [Ros95] presents a dynamic linking scheme for Nemesis. Each invocation of a library modules has its own *state buffer*, containing private static data. Modules are accessed via *interface references*, which are instantiated from *module interfaces* when resolving the symbolic reference to the module. An interface reference points to a structure containing a pointer to an *operations table* and the state buffer. A function is invoked by an indirect jump via the operations table, and the interface reference is passed as a parameter. As a result, three memory reads are required to find the address of the function to be called.

4 Linking in Mungi

Mungi supports static linking (by copying) as well as a version of dynamic linking, designed to retain the full flexibility dynamic linking offers in Unix system while minimising run-time overheads.

During execution of a program in Mungi, a *data segment* containing private static variables is associated with every instantiation of a dynamically linked module. While executing such a module's code, its data segment's address is held in the *data segment register*.[4] The data segment also contains *module descriptors* of imported modules. A module descriptor contains pointers to all functions imported from the module, plus a pointer to the data segment of the exporting module. This is shown in Figure 1.

4.1 Initialisation of module descriptors

Module descriptors are allocated in the importing module's data segment by the linker. To initialise a module descriptor at run time, the importing module calls the exporting module's *constructor*, passing the address of the descriptor as a parameter. In order to avoid multiple instantiation of modules which are imported by several other modules (such as `libc` in Figure 1), the constructor is also passed a pointer to a table of already instantiated modules. This table is held in the main module's data segment.

After verifying that its module is not already instantiated, the constructor

[4]On the SGI Indy we use the *global pointer register* which Irix uses to point to the GOT, hence the number of registers available to the compiler does not change with respect to Irix.

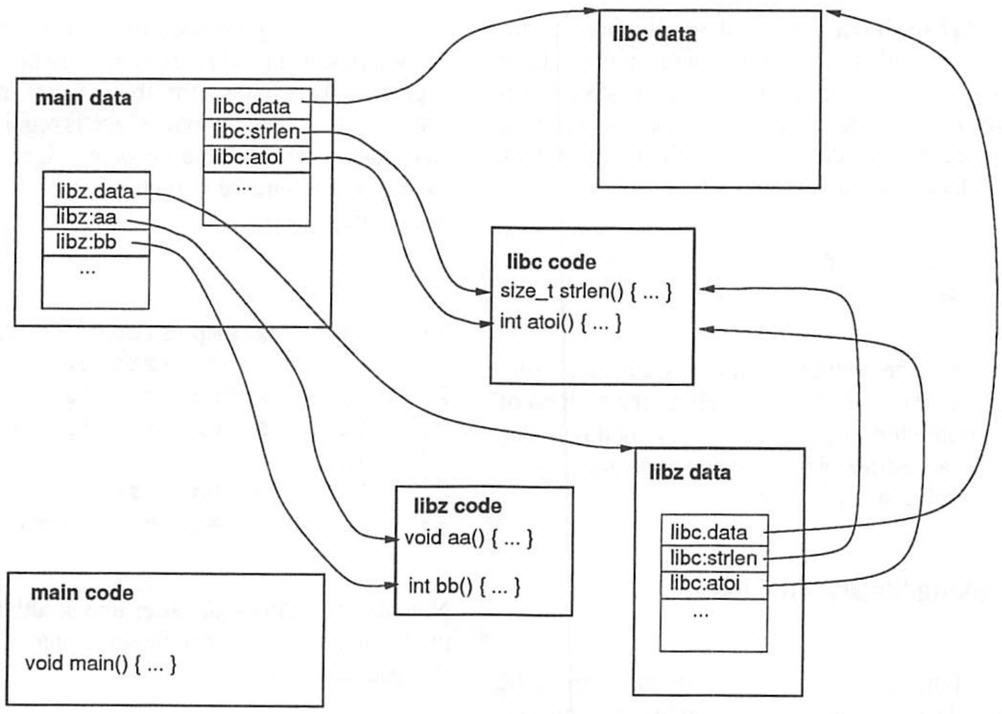


Figure 1: Memory objects (bold boxes) and module descriptors (other boxes) during execution of a dynamically linked Mungi program.

- allocates and initialises a new data segment,
- initialises the module descriptor passed by the caller, and
- updates the table of instantiated modules.

Only the second step needs to be performed for a module which has already been instantiated, in which case the address of the module's data segment is obtained from the table of instantiated modules.

The constructor is passed a third parameter, the expected size of the module descriptor. The constructor will only initialise the descriptor up to this specified size. This supports evolution of library modules — further entrypoints can be added to a library without breaking existing applications, provided that new entrypoints are added at the end.

Initialisation of the main module is somewhat different. Like a constructor, the startup code needs to allocate and initialise a data segment. In addition, the startup code must initialise the table of already instantiated modules. There is no problem with a module having startup code as well as a constructor, such a module can then be imported by other modules as well as being executed as a program.

4.2 Lazy initialisation

The first step above, allocation and initialisation of the new data segment, involves calling the constructors of all imported modules. To avoid the obvious recursion problem with cyclic reference, the constructor must at this stage mark its module's entry in the table of instantiated modules as partially initialised.

Alternatively, modules can be instantiated lazily, in analogy to "lazy loading" of library modules in Unix systems.[5] As lazy loading in Unix, this reduces task startup cost and reduces total overhead if a module is linked but not actually accessed at run time (at the cost of delaying fatal "module not found" errors until well into the execution).

[5]While there is some similarity to lazy loading, it is important to note that there is no explicit "loading" step in a SASOS — everything is already in virtual memory, and is made resident by the demand paging system on access. As far as physical memory is concerned, lazy loading is normal in a SASOS, but does not have the same drawback of delaying irrecoverable errors when libraries are removed.

A lazily initialised module has its descriptor point to initialisation stubs rather than module entry points. Each stub calls the lazy initialisation routine (which is statically linked to the module), passing it an index to its own position in the module descriptor. On the MIPS R4600, such a stub looks as follows:

```
1:   li    $reg, constIndex
2:   b     lazyInitialiser
```

The initialiser, after setting up the module's data segment, replaces the pointer to the stub by the address of the appropriate entrypoint in the library module. The stubs require an extra 8 bytes of space per entrypoint — really a negligible space overhead.

4.3 Invoking library functions

To invoke a function called `printf` imported from the library module `libc`, the following code is executed on the MIPS R4600:

```
1:   ld     $temp,libc_descr+\
                  printf_index($dseg)
2:   sd     $dseg,const($sp)
3:   ld     $dseg,libc_descr($dseg)
4:   jalr   $temp
5:   ld     $dseg,const($sp)
```

The first line loads the address of the `printf` function into a temporary register. This address (relative to the beginning of the data segment) is determined by the linker, and is at a constant offset from the data segment register. The next line saves the data segment register of the calling module on the stack, and line 3 sets up the segment register for the called module. Line 4 invokes the library function and line 5 restores the data segment register after its return. This code executes in 5 cycles on the R4600 (single-issue) CPU.

Note that on the R4600, a jump to a constant immediate 64-bit address (as would be used in a naive implementation of static linking) takes 7 cycles. Irix reduces this to 2–3 cycles (depending on the ability to make use of load delay slots) by using a *global pointer register* pointing to a table of entry point addresses. Comparing this with the 5 cycles required to call a dynamically linked function in Mungi indicates that the run-time overhead of dynamic linking in Mungi is only an additional 2–3 cycles per call of an imported function. This is a very small overhead.

The above invocation code only works if the `printf` entry is less than 64kB from the beginning of the data segment. This allows for about 8,000 imported functions (actually less, as some space is required for private static data). If the table becomes bigger, a somewhat longer code sequence is required, which takes 7 cycles to execute:

```
1:   ld     $temp,libc_interf_ptr
                   ($dseg)
2:   sd     $dseg,const($sp)
3:   ld     $tmp2,printf_index($temp)
4:   jalr   $tmp2
5:   ld     $dseg,0($temp)
6:   ld     $dseg,const($sp)
```

Note that the CPU stalls after line 3, although the compiler may be able to schedule some other instruction into the load delay slot.

We found that the biggest libraries generally used in Unix systems, `libc` and `libX11`, have between 1,000 and 1,500 entry points. (Many of these are actually internal and would not be exported if the C language provided better control over export of functions from libraries. Mungi's *module descriptors*, described in Section 5 provide such control and will therefore result in smaller interfaces for the same functionality.) We therefore expect that the shorter code sequence will almost always suffice.

5 Discussion

One remaining issue is that of modules exporting variables. For example, POSIX [POS90] specifies that the global variable `errno` is used to inform clients of the reason for the failure of an operation. This cannot be supported by Mungi's dynamic linking scheme.

It is, of course, always possible to avoid this problem by resorting to static linking — a highly unsatisfactory way out. However, exporting global variables from library modules is very bad practice, as it is not multi-threading safe. For that reason, Unix systems must break POSIX compliance if they want to support multithreaded processes. Modern Unix systems inevitably[6] use a construct like

[6] We checked Irix, Digital Unix, Solaris and Linux.

```
extern int *__errno();
#define errno (*(__errno()))
```

to declare errno when multithreading is supported. The same works in Mungi without problems.

Another issue concerns function pointers, which are used, for example, by the C qsort() utility and to implement virtual functions in C++. Function pointers can no longer be simply entrypoint addresses, as invoking a function requires loading the data segment register prior to branching to the entry point. Hence a "function pointer" must consist of an (address, global pointer) pair. This does not cause problems with portability of properly written application code, as the C standard [ISO90] makes no assumption about the size of function pointers and explicitly prohibits casts between function pointers and other pointers. Unfortunately, most compilers do not enforce this rule and, consequently, there exists plenty of non-conforming code. However, "bug-compatibility" is not a design goal of Mungi, and we therefore do not consider this a significant problem. The format change of function pointers is the only change required to standard Unix compilers to allow them to support Mungi's dynamic linking scheme.

More changes are required for linking. We decided not much was to be gained by porting a Unix linker, and instead implemented a Mungi linker from scratch. Its size is about 4,000 lines of C.

The mechanics of preparing code for execution differs somewhat between Mungi and Unix. The main reason for this is the need to generate a different calling sequence for functions exported by dynamically linked libraries. In order to do this, the assembler must know which entrypoints will be loaded from a dynamic library.

This is supported by a *module description object* (MDO) associated with each library module. The MDO is a simple text object (which is presently created manually, although tools will automate this in the near future). It contains a list of entry points exported by the module, and a list of imported modules. Figure 2 gives examples of module descriptions.

C source objects are presently compiled to assembly language by an unmodified GNU C compiler.[7] The gcc output is then processed by the GNU assembler, which we modified to generate the proper calling sequence for cross-module calls and to access private static data from

```
libc.mm                 main.mm
[IMPORTS]               [IMPORTS]
                        libc.mm
[EXPORTS]               libz.mm
strlen
atoi                    [EXPORTS]
...
                        [OBJECTS]
[OBJECTS]               main.o
c1.o                    sub.o
c2.o                    ...
...
```

Figure 2: Sample module descriptions: At the left, a typical description (libc.md) of a library module libc.mm is given, while at the right the module description (main.md) of a program module main.mm is shown. Names correspond to Figure 3.

the module's data segment. The assembler reads the MDO in order to identify cross-module calls and produces relocatable binary objects.

When creating a library, the Mungi linker is used to (statically) link all of the library's relocatable objects into a single library module object; the linker determines the exported entry point from the MDO. It also adds the initialisation code for the library module, as well as the initialisation stubs which invoke it and the module constructor.

When preparing an executable module, the linker reads the MDOs of all imported libraries and creates the appropriate initialisation stubs for all imported functions, and (statically) links all remaining relocatable modules into the new program module, which is then ready for execution. Unlike Unix, no "run-time linker/loader" is required, as each module has its own initialisation code. The mechanics of Mungi linking are shown in Figure 3.

The Macintosh on the PowerPC uses a similar approach to dynamic linking [App94]. Function references in MacOS are represented by a "transition vector", which consists of the entrypoint address and the address of a "table of contents" (TOC), essentially the module's data segment.[8] The TOC contains pointers to imported functions, in the form of transition vector addresses, as well as pointers to the module's static data. C language function pointers are also represented as transition vector addresses.

[7]As indicated a few paragraphs earlier, this implies that it is presently not possible to invoke function arguments outside their own module without modifications to the C source.

[8]The Macintosh terminology for modules is "fragments" but in order to avoid a proliferation of terms we will stick to calling them "modules".

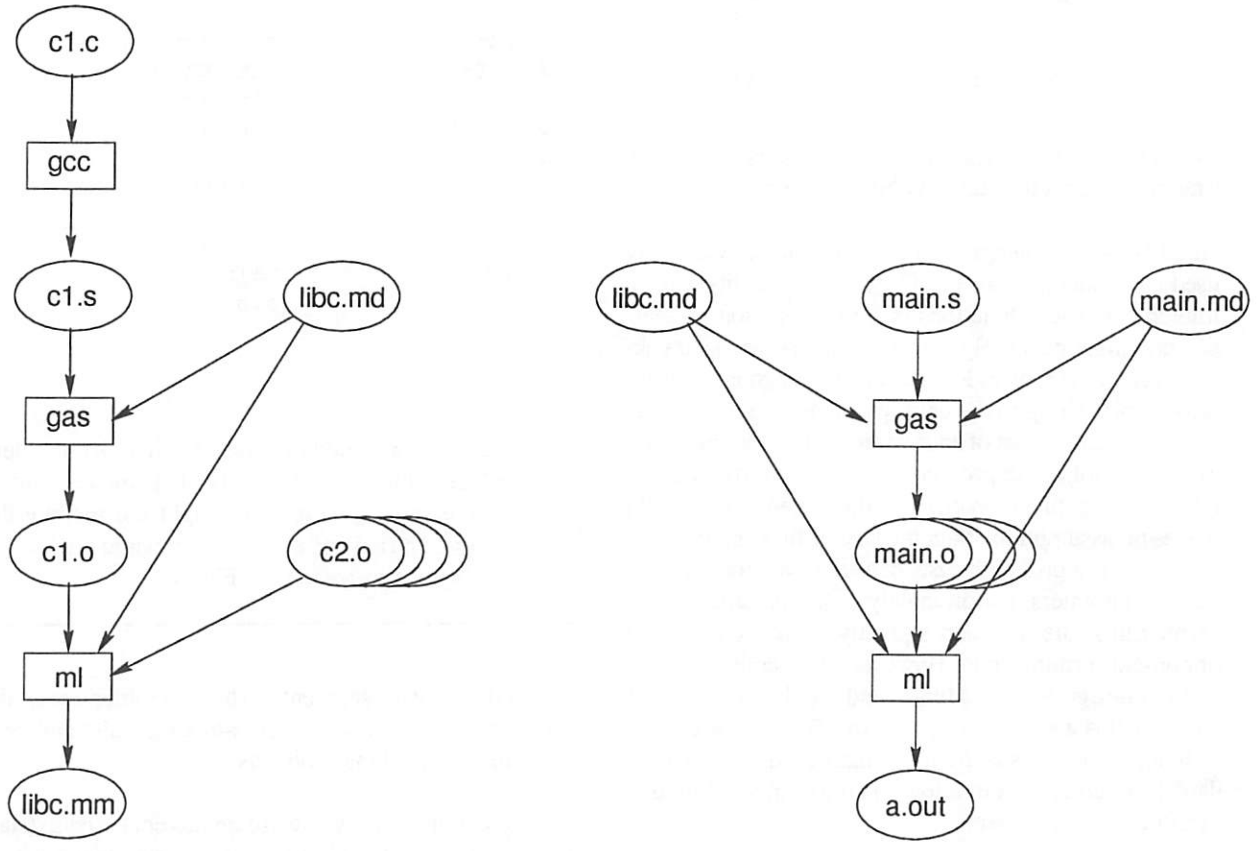


Figure 3: Dataflow for linking in Mungi. Left: A source object `c1.c` is compiled and linked with other relocatables into a dynamically linkable library object `libc.mm`. Right: A program `main.c` is compiled and linked into an executable module `a.out`.

The present module's TOC address is contained in the "table of contents register" (RTOC, equivalent to our data segment register). The invocation sequence for imported functions uses a double indirection. The caller loads the RTOC with the address of the callee's transition vector (i.e., the contents of the function pointer). The callee then loads the RTOC with the new TOC address by an indirect load through the present RTOC value. The main difference to our scheme is the extra indirection.

While the MacOS scheme is similar to ours, this does not obviate the need for position-independent code (called "pure executable code") on the Macintosh. This is because MacOS is not a SASOS, and can therefore not ensure that a dynamic library module is always linked at the same address.

6 Performance

Performance figures are shown in Table 1. The table gives execution times of a benchmark program (OO1 [CS92] run as a single process as in [HEV$^+$98]) for static and dynamic linking under Irix 6.2 and Mungi. All runs were performed on the same hardware running either Irix or Mungi in single-user mode. Lazy loading (for Irix) and lazy initialisation (for Mungi) were turned off to make timings more consistent. (As explained earlier, lazy loading/initialisation does not normally reduce overall runtime, only start-up latency.) Irix runs used the Irix 6.2 C compiler, assembler and linker, while Mungi runs used GCC version 2.8.1, the GNU assembler version 2.8.1 (modified to support dynamic linking) and our linker.

OO1 tests operations which are considered typical for object-oriented databases: lookup and insertion of objects, and traversal of inter-object references. The specification of OO1 requires a database server running on a

	Irix/32-bit/SGI-cc			*Mungi/64-bit/GCC*		
	static	*dynamic*	*dyn/stat*	*static*	*dynamic*	*dyn/stat*
lookup	7.26(3)	8.02(3)	1.104(10)	7.568(6)	8.199(4)	1.083(3)
forward traverse	4.77(3)	5.17(4)	1.084(15)	6.013(6)	6.040(3)	1.004(6)
backward traverse	5.13(2)	5.68(4)	1.107(12)	6.976(4)	7.011(4)	1.005(1)
insert	4.61(2)	5.02(2)	1.087(10)	4.528(4)	4.755(3)	1.051(1)
total	21.7(1)	23.9(1)	1.097(12)	25.08(1)	26.00(1)	1.037(1)

Table 1: Average execution times in ms of OO1 operations (single-process version, see [HEV$^+$98]). Figures in parentheses indicate standard deviations in units of the last digit. Irix figures are for 32-bit execution using the SGI C compiler, while Mungi figures are for 64-bit executing using GCC.

machine different from the client, and also specifies that all updates are to be flushed to disk at certain points. As we are here only interested in the cost of function invocation, we ignored that part of the specification and ran everything in memory, without any I/O, and in a single process. In every other respect we followed the OO1 specification.

We implemented the "server part" of the database in a library module which is invoked by application code via normal function calls. In line with the OO1 specification, server invocations pass a function pointer to the database which the database invokes to obtain further data or pass data back to the client. The *lookup* part of the benchmark consists of 1000 server invocations, each looking up a different object, and each call invoking a client function passed as a parameter. The *forward traversal* operation consists of a single invocation of the server code, which invokes a client procedure 3,280 times (for different objects directly or indirectly linked to the object referenced in the server invocation). *Backward traversal* is similar; the actual invocation counts are different but, in average, the same as for forward traversal. The *insert* benchmark consists of 100 calls to the database, each inserting a new object into the database and in the process calling a client procedure three times. Hence the total benchmark performs about 8960 cross-module calls. The *lookup* and *insert* operations mainly exercise the index data structure (a B$^+$ tree in our implementation) while the *traversal* operations mostly follow internal links and thus perform mostly random accesses to the data without much use of the index structure.

This benchmark was selected as it is "tough" in the sense that it is dominated by cross-module calls to functions performing very little work. As it is the cross-module tasks which bear the dynamic linking overhead in Mungi, this test stresses the overheads of the SASOS dynamic linking scheme. A benchmark consisting of the same number of function calls, with a larger fraction of calls being inter-module, would reduce the total overheads in Mungi while leaving Irix' overheads unchanged.

The table shows that on Irix there is an average 10 % penalty for using dynamically linked libraries, while on Mungi the penalty is less than 4 %. The average overhead due to dynamic linking of an inter-module function invocation in Mungi comes to about $0.1\mu s$ or about ten cycles on the R4600. This is more than the number of extra instructions required in the calling sequence for dynamically linked code. The difference can be explained with an increased number of cache misses.

The significantly lower overhead of dynamic linking in Mungi as compared to Irix is mostly due to the fact that the Mungi scheme does not require position-independent code. The somewhat higher overhead of inter-module function invocation in Mungi (2–3 cycles more than in Irix) is more than compensated by not requiring position-independent code. Furthermore, the overhead in the Mungi scheme only applies to inter-module invocations, where in Irix inter-module calls to exported functions have the same overhead.

We also attempted to measure the initial overhead of dynamic libraries, i.e. the invocation overhead of the constructor which initialises the data segment. However, this overhead is so small that we could not measure it reliably for either Irix or Mungi. In both cases it is at most a few tens of microseconds.

While Mungi has a significantly lower penalty for dynamic linking than Irix, a seemingly disturbing observation from Table 1 is that code seems to execute generally slower under Mungi than under Irix. However, it must be kept in mind that all Mungi executions are true 64-bit, while Irix only supports 32-bit execution on the Indy. 32-bit code is inherently faster on the MIPS

	Irix	*Mungi*		*Mungi/Irix*	
		good	*bad*	*good*	*bad*
lookup	7.367	7.169	7.452	0.973	1.012
forward traverse	5.904	6.085	6.079	1.031	1.030
backward traverse	6.796	6.992	6.991	1.029	1.029
insert	4.755	4.724	4.801	0.993	1.010
total	24.822	24.970	25.323	1.006	1.020

Table 2: Average execution times in ms of OO1 operations on for 64-bit execution of statically linked code on Irix and Mungi. Code is compiled with GCC and assembled with the SGI assembler and finally linked with the native (SGI or Mungi) linker. "Good" vs "bad" in the Mungi numbers refers to the cache friendliness of the stack alignment and the last two column give Mungi execution time normalised to Irix times.

R4600 as loading a constant address requires more cycles for 64-bit than for 32-bit addresses. The tests were also run with different compilers: Mungi benchmarks could only be compiled with GCC, as SGI's C compiler/assembler/linker toolchain does not support our dynamic linking scheme, while the GNU assembler and linker do not support Irix. Finally, different implementations of the `strcpy()` C library functions are used.

In order to eliminate the effects of 32-bit vs. 64-bit execution and differing tool chains and C libraries, we did a direct comparison, running an identically compiled version of the statically linked benchmark code in 64-bit mode on both systems. This meant compiling and assembling the code, including the C library, using GCC and the SGI assembler, and then linking it for Irix with the SGI linker, and for Mungi with our linker. As the benchmarks only time user code (no system calls are performed between time stamps, and the timer overhead is subtracted), this means that identical instructions are executed on both systems.

As Irix does not support 64-bit code on our platform, we had to patch the executable to pretend to the loader that it was a 32-bit image. This approach works under certain circumstances (as long as only a very limited set of system calls are used), but only for statically linked code. One required system call where extra work was required is `gettimeofday()`: As the format of the `timeval` struct differs between the 32-bit and the 64-bit Irix APIs, we had to use the 32-bit C library interface for this call.

In order to verify that these modifications do not affect performance of the Irix executable, we ran the "proper" 64-bit image as well as the patched one on an SGI machine supporting 64-bit executions. We found that the execution times of the two versions were identical.

The results of running the same code in 64-bit mode on Irix and Mungi are shown in Table 2. Two sets of Mungi results are presented: "good" and "bad", which differ only in the address at which the user stack is allocated. The stack address affects the results as it affects conflict misses in the Indy's data cache. The R4600 features separate on-chip instruction and data caches, both 2-way set associative and 16kB big [R4k95]. The Indy does not have secondary caches and thus has a high cache-miss penalty. The Mungi execution times recorded as "good" and "bad" in Table 2 correspond to the most and least cache friendly stack layout, respectively. They differ by about 1.5 %, which gives an indication of the impact of cache effects on the results. Irix runs used the default layout (which is cache friendly).

Comparing the Irix times with the "good" Mungi times it can be seen that they are very close. Mungi is between one and three percent faster on *lookup* and *insert*, and about three percent slower on the *traverse* benchmarks. For the total benchmark time these almost average out, with Mungi being 0.6 % slower. Given the fact that Mungi is several percent faster on some benchmarks and Irix on others, that overall difference is negligible and insignificant. They are much smaller than the performance gain of Mungi's dynamic linking scheme compared to the one used in Irix.

It is nevertheless interesting to speculate about the sources of these remaining differences. We can think of two possible reasons for the observed discrepancies in execution times: TLB misses and other cache effects.

The R4600 has a software-loaded, fully associative, 48-entry tagged TLB; each entry maps a pair of virtual pages [R4k95]. Hence the TLB can map a maximum of 96 pages, or 384kB. As the total database is about 4MB in size, and the benchmark is designed to access its contents randomly, a significant number of TLB misses is expected, particularly in the *traverse* operations.

Mungi is implemented on top of the L4 microkernel [EHL97], hence TLB misses are handled by L4. The microkernel's TLB miss handler is highly optimised and loads the TLB from a software cache [BKW94, EHL99] which is big enough to hold all page table entries required for the benchmark. However, the need to support 64-bit address spaces makes L4's TLB miss handler inherently slower than what can be achieved in a system only supporting 32-bit address spaces. Slightly slower handling of TLB misses in L4, and thus Mungi, is a likely explanation for the somewhat slower Mungi execution in the *traverse* benchmarks (which particularly exercise the TLB).

Other cache effects which could impact on the results are instruction cache conflicts. While we made certain that the same user-mode instructions are executed in both benchmarks, the layout of the executable still differs as a result of linking different system libraries and the linkers using different strategies for collecting relocatable modules. These differences can lead to different cache miss patterns. The *traverse* benchmarks contain the largest number of cross-module invocations (and hence non-local jumps) and are most likely to be affected.

7 Conclusions

In this paper we have reviewed linking in a Unix system and examined the issues relating to linking in a single address space system. We have presented a dynamic linking scheme for Mungi and have discussed its merits and limitations. Benchmarking shows that the run-time overhead of Mungi's dynamic linking scheme is less than half of dynamic linking in Irix, in a scenario which favours Irix.

The performance advantages of the Mungi dynamic linking scheme could also be obtained in traditional systems on 64-bit architectures *if they used a global address space for dynamically-linked libraries*. As in *quickstart*, a region of the address space must be reserved for library modules, and each participating module must be linked at the same address in all processes. Such a scheme can eliminate the need for position independent code even in traditional systems. It requires a system-wide manager which hands out unique address regions for linking libraries. Each of a participating library's clients must follow the protocol of always linking the library at this same address. A single-address-space operating system guarantees this automatically; in such a system *every* object is always mapped to a fixed virtual memory address.

8 Acknowledgements

Luke Deller gratefully acknowledges his School of Computer Science & Engineering Vacation Scholarship under which most of the work presented here was performed. The project also received support from the Australian Research Council under the Small Grants Scheme.

We appreciate the helpful suggestions and comments from our shepherd Chris Small and from anonymous USENIX reviewers.

9 Availability

Mungi will be freely available in source form under the GNU General Public License from http://www.cse.unsw.edu.au/~disy/Mungi.html.

References

[App94] Apple Computer Inc. *Inside Macintosh: PowerPC System Software*. Addison-Wesley, 1994.

[Ber80] Viktors Berstis. Security and protection in the IBM System/38. In *Proceedings of the 7th Symposium on Computer Architecture*, pages 245–250. ACM/IEEE, May 1980.

[BKW94] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 243–253, Monterey, CA, USA, 1994. USENIX/ACM/IEEE.

[Cha95] Jeffrey S. Chase. *An Operating System Structure for Wide-Address Architectures*. PhD thesis, University of Washington, 1995. URL http://www.cs.duke.edu/-chase/research/thesis.ps.

[CLBHL92] Jeff S. Chase, Hank M. Levy, Michael Baker-Harvey, and Edward D. Lazowska. How to use a 64-bit virtual address space. Technical Report 92-03-02, Department of

Computer Science and Engineering, University of Washington, Seattle, WA, USA, 1992. URL ftp://ftp.cs.washington.edu/tr/-1992/03/UW-CSE-92-03-02.PS.Z.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12:271–307, November 1994.

[CS92] R. G. G. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database Systems*, 17:1–31, 1992.

[DEC94] Digital Equipment Corp. *DEC OSF/1 Programmer's Guide*, 1994. Order No AA-PS30C-TE.

[EHL97] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. *L4 Reference Manual — MIPS R4x00.* School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, December 1997. UNSW-CSE-TR-9709. Latest version available from http://www.cse.unsw.edu.au/~disy/.

[EHL99] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. Page tables for 64-bit computer systems. In *Proceedings of the 4th Australasian Computer Architecture Conference*, pages 211–226, Auckland, New Zealand, January 1999. Springer Verlag. Available from URL http://www.cse.unsw.edu.au/~disy/.

[HEV+98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[ISO90] *International Standard, ISO/IEC 9899, Programming Languages — C*, 1990.

[POS90] *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]*, 1990. IEEE Std 1003.1-1990, ISO/IEC 9945-1:1990.

[R4k95] Integrated Device Technology. *IDT79R4600 and IDT79R4700 RISC Processor Hardware User's Manual*, April 1995.

[Ros95] Timothy Roscoe. *The Structure of a Multi-Service Operating System.* Phd thesis, University of Cambridge Computer Laboratory, April 1995. TR-376, URL http://www.cl.cam.ac.uk/ftp/papers/-reports/TR376-tr-multi-service-os.ps.gz.

[RSE+92] Stephen Russell, Alan Skea, Kevin Elphinstone, Gernot Heiser, Keith Burston, Ian Gorton, and Graham Hellestrand. Distribution + persistence = global virtual memory. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 96–99, Dourdan, France, September 1992. IEEE.

[Sol96] Frank G. Soltis. *Inside the AS/400.* Duke Press, Loveland, CO, USA, 1996.

[WM96] Tim Wilkinson and Kevin Murray. Evaluation of a distributed single address space operating system. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 494–501, Hong Kong, May 1996. IEEE.

[WSO+92] Tim Wilkinson, Tom Stiemerling, Peter E. Osmon, Ashley Saulsbury, and Paul Kelly. Angel: A proposed multiprocessor operating system kernel. In *European Workshop on Parallel Computing*, pages 316–319, Barcelona, Spain, 1992.

[X/O90] X/Open. *System V Application Binary Interface*, 3.1 edition, 1990.

The Design and Implementation of a DCD Device Driver for Unix *

Tycho Nightingale†, Yiming Hu‡, and Qing Yang†

† *Dept. of Electrical & Computer Engineering*
University of Rhode Island
Kingston, RI 02881
{tycho,qyang}@ele.uri.edu

‡ *Dept. of Ele. & Comp. Eng. and Comp. Sci.*
University of Cincinnati
Cincinnati, OH 45221
yhu@ececs.uc.edu

Abstract

Recent research results [1, 2] using simulation have demonstrated that *Disk Caching Disk* (*DCD*), a new disk I/O architecture, has the potential for drastically improving disk write performance besides its higher reliability than traditional disk systems. To validate whether DCD can live up to its promise in the real world environment, we have designed and implemented a DCD device driver for the Sun's Solaris operating system. Measured performance results are very promising. For metadata intensive benchmarks, our DCD driver outperforms the traditional system by a factor of 2–6 in terms of program execution speeds. The driver also guarantees file system integrity in the events of system crashes or failures. Moreover, unlike other approaches such as Log-structured File Systems or Soft Updates, the DCD driver is completely transparent to the OS. It does not require any changes to the OS or the on-disk data layout. As a result, it can be used as a "drop-in" replacement for the traditional disk device driver in an existing system to obtain immediate performance improvement. Our *multi-layered device-driver* approach significantly reduces the implementation overhead and improves portability.

1 Introduction

Most Unix file systems use synchronous writes for metadata updates to ensure proper ordering of changes being written into disks [3, 4]. The orderly updating is necessary to ensure file system recoverability and data integrity after system failures. However, it is well known that using synchronous writes is very expensive because it forces the updates to be processed at disk speeds rather than processor/memory speeds [3]. As a result, synchronous writes significantly limit the overall system performance.

Many techniques have been proposed to speed up synchronous writes. Among them the *Log-structured File System* (LFS) [5, 4, 6] and *metadata logging* [7] are two well-known examples. In LFS, the only data structure on the disk is a log. All writes are first buffered in a RAM segment and logged to the disk when the segment is full. Since LFS converts many small writes into a few large log writes, the overhead associated with disk seeking and rotational latency is significantly reduced. In metadata logging, only the changes of metadata are logged, the on-disk data structure is left unchanged so the implementation is simpler than that of LFS. Both LFS and metadata logging provide higher performance and quicker crash-recovery ability than traditional file systems.

Soft Updates [3] converts the synchronous writes of metadata to delayed writes to achieve high performance. Soft Updates maintains dependency information of metadata in its kernel memory structures. The dependency information is used when an updated dirty block is flushed to a disk to make sure that data in the disk is in a consistent state. Ganger and Patt have shown that Soft Updates can significantly improve the performance of benchmarks that are metadata update intensive, when compared to the conventional approaches. Soft Updates requires only moderate changes to the OS kernel (Ganger and Patt's implementation on the SVR4 Unix MP system consists of 1500 lines of C code). In addition, it does not require changes to the on-disk data structures.

LFS, metadata logging, and Soft Updates all have good performance. However, they are not without limitations. For example, LFS and metadata logging require redesigning of a significant portion of the file system. Soft Updates also needs some modifications to the OS kernel source code which is not easily accessible to many researchers. Partly due to these limitations, LFS, metadata logging and Soft Updates

*This research is supported in part by National Science Foundation under Grants MIP-9505601 and MIP-9714370.

are not available on many commonly used Unix systems. Moreover, the implementations of LFS, metadata logging and Soft Updates are intrinsically tied to kernel data structures such as the buffer cache and the virtual memory system. They often become broken and require re-implementations when the kernel internal data-structures are changed with OS upgrading. For example, Seltzer et al. implemented an LFS for 4.4BSD [4]; McKusick implemented Soft Updates for FreeBSD, a descendant of the 4.4BSD system. Both the LFS and the Soft Updates code have been broken as a result of the evolution of FreeBSD. While the Soft Updates code is again working in FreeBSD 3.1, the latest stable version as of this writing, the LFS code is not. In fact, the FreeBSD team has decided to drop the effort to fix the LFS code because of the lack of human resources, and the LFS code has even been removed from the FreeBSD CVS repository.

In this research we used another approach that avoids many of the above limitations. Our approach is based on a new hierarchical disk I/O architecture called *DCD (Disk Caching Disk)* that we have recently invented [1, 2, 8]. DCD converts small requests into large ones before writing data to the disk. A similar approach has been successfully used in database systems [9]. Simulation results show that DCD has the potential for drastically improving write performance (for both synchronous and asynchronous writes) with very low additional cost. We have designed and implemented a DCD device driver for Sun's Solaris operating system. Measured performance results show that the DCD driver runs dramatically faster than the traditional system while ensuring file system integrity in the events of system crashes. For benchmarks that are metadata update intensive, our DCD driver outperforms the traditional system by a factor of 2–6 in terms of program execution speeds. Moreover, the DCD driver is completely transparent to the OS. It does not require any changes to the OS. It does not need accesses to the kernel source code. And it does not change the on-disk data layout. As a result, it can be used as a "drop-in" replacement of the traditional disk device driver in an existing system to obtain immediate performance improvement.

The paper is organized as follows. The next section presents the overview of the DCD concept and the system architecture, followed by the detailed descriptions of the design and operations of the DCD device driver in Section 3. Section 4 discusses our benchmark programs and the measured results. We conclude the paper in Section 5.

2 Disk Caching Disks

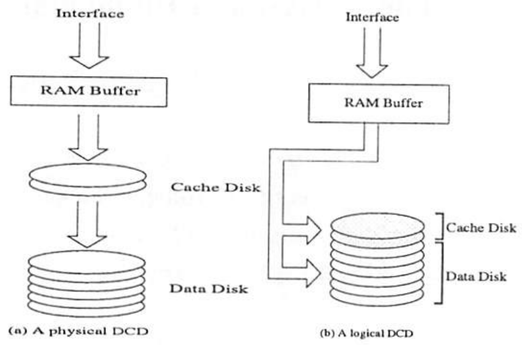


Figure 1: The structure of DCD

Figure 1 shows the general structures of DCD. The fundamental idea behind DCD is to use a small log disk, called *cache-disk*, as an extension of a small NVRAM (Non-Volatile RAM) buffer on top of a *data-disk* where a normal file system exists. The NVRAM buffer and the cache-disk together form a two-level cache hierarchy to cache write data. Small write requests are first collected in the small NVRAM buffer. When the NVRAM buffer becomes full, the DCD writes all the data blocks in the RAM buffer, *in one large data transfer*, into the cache-disk. This large write finishes quickly since it requires only one seek instead of tens of seeks. As a result, the NVRAM buffer is very quickly made available again to accept new incoming requests. The two-level cache appears to the host as a large virtual NVRAM cache with a size close to the size of the cache-disk. When the data-disk is idle or less busy, it performs *destaging* operations which transfer data from the cache-disk to the data-disk. The destaging overhead is quite low because most data in the cache-disk are short-lived and are quickly overwritten, therefore requiring no destaging at all. Moreover, many systems, such as those used in office/engineering environments, have sufficient long idle periods between bursts of requests. For example, Blackwell et al. [10] found that 97% of all LFS cleaning could be done in the background on the most heavily loaded system they studied. Similarly, DCD destaging can be performed in the idle periods, therefore will not interfere with normal user operations at all.

Since the cache in DCD is a disk with a capacity much larger than a normal NVRAM cache, it can achieve very high performance with much less cost. It is also non-volatile and thus highly reliable.

The cache-disk in DCD can be a separate physical disk drive to achieve high performance (a *physical*

DCD) as shown in Figure 1(a), or one logical disk partition physically residing on one disk drive for cost effectiveness (a *logical DCD*) as shown in Figure 1(b). In this paper we concentrate on implementing the logical DCD, which does not need a dedicated cache-disk. Rather, a partition of the existing data-disk is used as a cache-disk. In addition, the RAM buffer shown in Figure 1(b) is implemented using a small portion of the system RAM. Therefore this implementation is complete software without any extra hardware.

One very important fact is that the cache-disk of DCD is only a cache that is completely transparent to the file system. Hu and Yang pointed out that DCD can be implemented at the device or device driver level [1], which has many significant advantages. Since a device driver is well-isolated from the rest of the kernel, there is no need to modify the OS and no need to access the kernel source code. In addition, since the device driver does not access kernel data structures, it is less likely that any kernel changes will require a re-implementation of the driver. Furthermore, since DCD is transparent to the file system and does not require any changes to the on-disk data layout, it can easily replace a current driver without affecting the data already stored on the disk (as long as a free cache-disk partition somewhere in the system can be provided). This is very important to many users with large amount of installed data.

DCD can speed up both synchronous writes and asynchronous writes. Although asynchronous writes normally do not have a direct impact on the system performance, they often affect the system performance indirectly. For example, most Unix systems flush dirty data in the file system cache to disks every 30–60 seconds for reliability reasons, creating a large burst of writes [1]. This large burst will interfere with the normal read and write activities of users. DCD can avoid this problem by using a large and inexpensive cache-disk to quickly absorb the large write burst, destaging the data to the data-disk only when the system is idle.

Using results of trace-driven simulations, Hu and Yang have demonstrated that DCD has superior performance compared to traditional disk architectures in terms of I/O response times [1]. However, recent studies [12] have shown that I/O response times are not always a good performance indicator in I/O systems because not all I/O requests affect the system behavior in the same way. Trace-driven simulations also have limited accuracy because there is no feedback between the traces and the simulators. The main objective of this research is to examine whether DCD will have good performance in the real world. We believe that the best way to evaluate the real world performance is to actually implement the DCD architecture and measure its performance in terms of program execution times, using some realistic benchmarks.

[1]Some systems use a rolling update policy [11] to avoid this problem.

3 Design and Implementation

This section describes the data structures and algorithms used to implement our DCD device driver. Please note that while we have adopted the DCD architecture from [1], our implementation algorithms are considerably different from the ones reported in [1]. We choose these new algorithms mostly because they offer better performance and simplified designs. In some cases a new algorithm was chosen because of the different design goals of the DCD device driver and the original DCD. For example, the original DCD uses an NVRAM buffer and it reorders data in the NVRAM buffer to obtain better performance. However, when implementing the DCD driver, we have to use the system DRAM as the buffer. We can not reorder data in the DRAM buffer before the data is written to the disk, otherwise we can not maintain the order of updates to the disk. Keeping the order has some impact on the performance, but it is the price that has to be paid for maintaining the file system integrity.

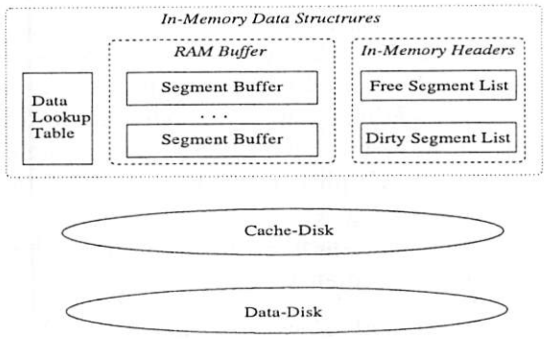


Figure 2: The Structure of the DCD Driver

Figure 2 shows the general structure of the DCD driver which consists of three main parts: the *in-memory data-structures*, a *cache-disk* and a *data-disk*. The In-memory data-structures include a *Data Lookup Table* which is used to locate data in the cache; a *RAM buffer* which comprises several *Segment Buffers* and a number of *In-memory Segment-Headers*. The *In-memory Segment-Headers* form two lists: the *Free Segment List* and the *Dirty Segment List.* The following subsections discuss these structures and the DCD operations in detail.

3.1 Cache-disk Organization

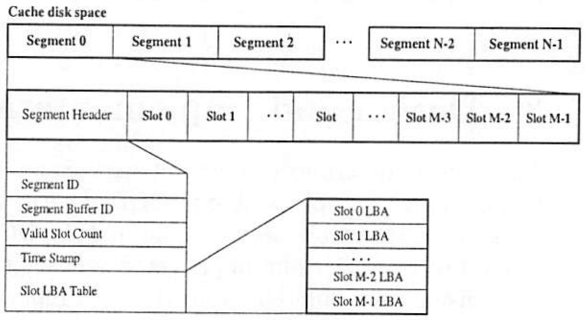


Figure 3: Cache-disk Organization

The original DCD in [1] writes variable-length logs into the cache-disk. However, when we tried to actually implement a DCD device driver, we found that variable-length logs make the destaging algorithm too complex, since it is difficult to locate a particular log in the cache disk. As a result, in the DCD device driver we use a fixed-length segment approach similar to the one used by LFS.

Figure 3 shows the data organization of the cache disk. The entire cache-disk is divided into a number of fixed-length *segments.* The data is written into the cache disk in fixed-size transfers equal to the segment-size, although the segment may be partially full. Each segment contains a *segment-header* followed by a fixed-number of data *slots.* A slot is the basic caching unit, and in our current implementation, can store 1 KB of data. The entire segment size is 128 KB plus two additional 512 byte blocks, used for the segment-header.

The segment-header describes the contents of a segment. It contains:

1. A *segment-ID* which is unique to each segment.
2. A *segment-buffer-ID* whose functions will be discussed later.
3. A *valid-slot-count* that indicates the number of slots in the segment containing valid data. This entry is used to speed up the destaging process.
4. A *time-stamp* which records the time when the segment is written into the cache-disk. During a crash recovery period, the time-stamps help the DCD driver to search for segments and to rebuild its in-memory data structures.
5. A *slot-LBA-table.* Each slot in the segment has a corresponding entry in the table. Each table entry is an integer that describes the *logical block address (LBA)* of the data stored in its slot. For example, if a data block written into block No. 12345 (i.e., its LBA) of the data-disk is currently cached in slot 1, then entry 1 of the table contains 12345 to indicate that slot 1 caches this particular data block. This information is needed since eventually data in the cache-disk will be destaged to their original addresses on the data-disk. If a slot in the segment is empty or contains invalid data, its entry in the slot-LBA-table is -1.

3.2 In-memory Segment-Headers

Once a segment is written into the cache-disk, its contents, including the segment-header (*on-disk segment-header*) are fixed. Furthermore, the on-disk segment-headers in the cache-disk are not accessed during the normal operations of the driver, but are used during crash-recovery. However, subsequent requests may overwrite data contained in the segment on the cache-disk, requiring updating of the segment-headers. For example, if a slot of an on-disk segment is overwritten, its corresponding *slot-LBA-entry* should be marked invalid, with a -1, to indicate that the slot no longer contains valid data. Additionally the *valid-slot-count* should be decreased by 1. It is prohibitively costly to update the on-disk segment-header for every such overwrite, therefore each segment-header has an in-memory copy (*in-memory segment-header*). Once a segment is written into the cache-disk, only the in-memory segment-header is updated upon overwriting. The on-disk segment-header remains untouched. If the system crashes, the in-memory segment-headers can be rebuilt by scanning the cache-disk and "replaying" the on-disk segment-headers in order of their time-stamps. Since the on-disk segment-headers have fixed locations on the disk, this scanning process can be done very quickly. We will discuss the crash recovery process in detail later in this section.

An in-memory segment-header is always in one of the following two lists: the *Free-Segment-List* and the *Dirty-Segment-List.* If a segment does not contain any valid data, its in-memory segment-header is in the Free-Segment-List. Otherwise it is put in the Dirty-Segment-List. When the system is idle, the destaging process picks up segments from the Dirty-Segment-List and moves valid data in these segments to the data disk. Once all data in a dirty segment is destaged, its header is moved to the Free-Segment-

List. Segments which are entirely overwritten during normal operation, i.e. the valid-slot-count becomes 0, are also put on the Free-Segment-List.

In our current prototype, the cache-disk is 64 MB and has about 512 segments (128 KB per segment). The size of each in-memory segment-header is 512 bytes. In total, 512 segment-headers require only 250 KB of RAM. Given today's low RAM cost, we put the entire 250 KB of headers in the system RAM so they can be accessed quickly.

3.3 Data Lookup Table

One of the most challenging tasks in this research is to design an efficient data structure and a search algorithm for the *Data Lookup Table*, used for locating data in the cache. In DCD, a previously written data block may exist in one of the following three places: the *RAM buffer*, the *cache-disk* and the *data-disk*. The Data Lookup Table is used to keep track of the location of data in these places. Since the driver must search the table for every incoming request, it is imperative to make the searching algorithm efficient.

In other I/O systems with a similar problem, a hash table with the LBA (Logical Block Address) of incoming requests as the search key, is usually used for this purpose. Unfortunately this simple approach does not work for the DCD driver. Since in Solaris and many other Unix systems, the requests sent to the disk device driver have variable lengths and may be overlapped. For example, suppose a data block B cached in the DCD has a starting LBA of S and a length of L. Another incoming request asks for a data block B' with a starting LBA of $S+1$ and a length of L'. If $L > 1$, then B and B' overlap, and the system should be able to detect the overlapping. Since B and B' have different LBAs, simply using the LBA of B' as the search key will not be able to find block B in the hash table.

Our profiling of I/O requests in Solaris demonstrated that the majority of requests are aligned to 4 KB or 8 KB boundaries and are not overlapped. Additionally, all requests were determined to be aligned to a 1 KB boundary. Therefore the Data Lookup Table was designed to efficiently handle the common cases (i.e., aligned un-overlapped 4 or 8 KB requests), while still able to handle other requests.

The minimum storage and addressing unit of the device driver was chosen to be a *cache fragment*, of size 1 KB and aligned to a 1 KB boundary. The fragments are designed to fit into the slots in the cache-disk segments. Since they are aligned and not overlapped they should be easy to find in the system. Although a fragment is the minimum unit used in the driver, most requests occupy several consecutive fragments. It is wasteful to break every incoming request into a series of fragments, and search the hash table for each fragment. Therefore a *cache line* was designed to hold 4 fragments or 4 KB of data aligned to a 4 KB boundary.

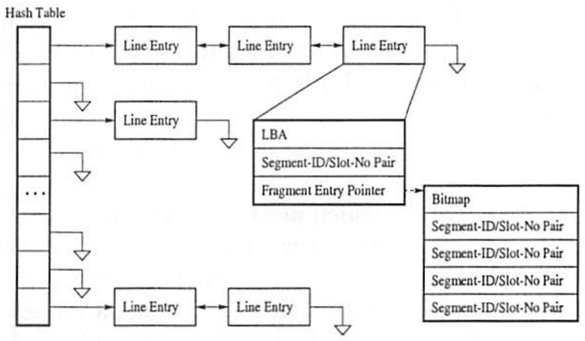


Figure 4: Data Lookup Table

The Data Lookup Table is in the form of a hash table chain, as is shown in Figure 4. Each entry in the Data Lookup Table, *line entry*, contains information associated with the blocks currently in the DCD. Each incoming request is mapped into one or more cache lines. The Data Lookup Table is then searched, using the LBA of the start of each cache line in the request as a search key. All entries in the hash table with the same hash value are linked together in a doubly-linked list.

A Data Lookup Table line entry consists of the following fields:

1. An *LBA* entry that is the LBA of the cache line.

2. A *segment-ID/slot-no* pair which uniquely determines the starting address of the data in the cache-disk. Although data cached in the DCD can be either in the RAM buffer or on the cache-disk, a unified addressing scheme is used. The segment-ID/slot-no pair are also used to address data in the RAM buffer. This unified scheme, discussed in more detail later, is used to reduce to bookkeeping overheard involved when data buffered in the RAM is written into the cache-disk.

3. An optional *fragment entry* pointer used to describe partial or fragmented cache entries.

While most I/O requests will map to one or two complete cache line entries, some requests are for *partial lines*. Also, overwriting can cause caches lines to be broken. For example, suppose a cache line has

4 consecutive fragments stored in segment S. If another request overwrites the second fragment in segment S and stores the new data in segment S', then the cache line becomes a *fragmented line* with data stored in several different places: the first fragment is in S; the second one in S'; and the remaining two fragments in S. To be able to handle both of these cases, the line entry, used in the Data Lookup Table, contains an optional pointer to a *fragment entry*, which describes both partial lines and fragmented lines.

Fragment entries contain the following fields:

1. A *bitmap* which marks the fragments of a cache line contain that valid data. Recall that a cacheline can contain 4 fragments, of 1 KB each, stored in 4 slots. If for example, a request contains only 2 KB, starting on a 4 KB boundary, then only the first two fragments of line are valid yielding a bitmap of 1100.

2. A table containing four possible *segment-ID/slot-no* pair entries which identify the locations of each fragment. The bitmap is used to determine which entries in the table are valid.

For every incoming request the Data Lookup Table is searched one or more times. If an entry is found in the Data Lookup Table (a cache hit), the requested block is contained somewhere in the DCD, either in the RAM buffer, or in the cache-disk. Otherwise if no corresponding entry is located (a cache miss), the requested block can be found on the data-disk.

Our I/O profiling results, of the prototype driver, show that most requests (50-90%) are contained in only one or two complete cache lines. The remaining requests need to access fragmented cache lines. Therefore the Data Lookup table is both space and time efficient.

3.4 Segment Buffers

Another important component of DCD is the RAM buffer. In our implementation the RAM buffer is divided into several (2–4) *Segment Buffers* which have the same structure as on-disk segments without segment-headers. Each Segment Buffer is assigned a unique *segment-buffer ID*.

Segment Buffers, much like their on-disk counterparts, are tracked using two lists: the *Free Segment-Buffer List* and the *Dirty Segment-Buffer List*, when not mapped to a segment. When a write request arrives, the driver picks a free segment buffer to become the *Current Segment-Buffer*. Meanwhile the driver also obtains a free disk segment from the Free Segment List to become the *Current Disk Segment*. The driver then "pairs" the RAM segment buffer and the disk segment together by writing the *segment-buffer ID* into the corresponding field into the header of the in-memory header of the disk segment.

From this point until the current Segment Buffer is written into the Current Disk Segment on the cache-disk, incoming write data is written into the slots of the segment buffer. However the Data Lookup Table will use the segment-ID/slot-no pairs of the Current Disk Segment as the addresses of these data (the unified addressing scheme). In addition, the RAM buffer does not have its own header. It shares the header of the disk segment. Imagine that the segment buffer "overlays" itself on top of the disk segment, as shown in Figure 5(a), therefore any data written to the slots of the Current Disk Segment is actually written into the slots of the Current Segment Buffer.

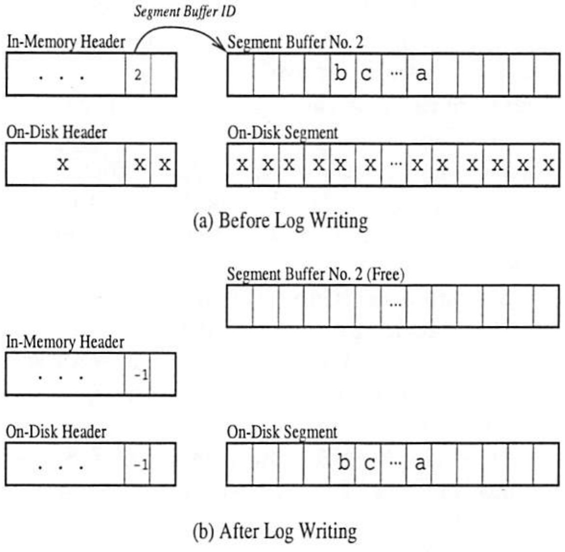


Figure 5: Segment Buffers and Disk Segments

If a read request tries to access a newly written data in the segment buffer before it is logged into the cache-disk, the Data Lookup Table will return a segment-ID/slot-no pair to indicate that the requested data is in a particular segment. However, the driver will know from the *segment-buffer ID* entry in the segment-header that the data are actually in the RAM buffer instead of in the on-disk segment.

When the current segment buffer becomes full, the driver schedules a log write so that the entire segment buffer is written into the Current Disk Segment on the cache-disk. Meanwhile another segment-buffer/disk-segment pair is chosen to accept new incoming requests. When the log write finishes, the driver then changes the segment-buffer ID field in the in-memory segment-header to -1 to indicate that all the data is now in the cache-disk, as shown in Figure 5(b). There is no need to update all of the Data Lookup Table entries for all the blocks in the segment. Finally the segment buffer is freed and placed on the Free Segment Buffer List, and the in-memory segment-header is placed on the Dirty Segment List waiting to be destaged.

3.5 Operations

3.5.1 Writes

After receiving a write request, the DCD driver first searches the Data Lookup Table and invalidates any entries that are overwritten by the incoming request. Next the driver checks to see if segment buffer/disk segment pair exists, if not, a Free Segment Buffer and Free Disk Segment are paired together. Then all of the data in the request is copied into the segment buffer, and their addresses are recorded as entries in the Data Lookup Table. Finally, the driver signals the kernel that the request is complete, even though the data has not been written into the disk. This *immediate-report* scheme does not cause any reliability or file system integrity problems, as will be discussed shortly in this section.

There are two cases where data written into the DCD is handled in a different manner. First, to reduce destage overhead, requests 64 KB or larger are written directly into the data-disk. Second, if a user request sets the "sync" flag in an I/O request, the request is also written directly into the data disk. Only after the requests finishes writing, does the driver signal completion to the kernel.

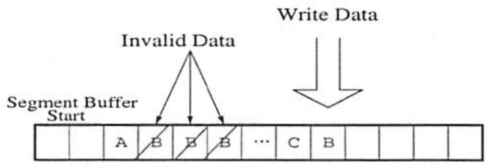


Figure 6: Overwriting in a Segment Buffer

When data is copied into the segment buffer, it is always "appended" to the previous data in the buffer, even when the new data overwrites previously written data. For example, Figure 6 shown a segment buffer containing blocks A, B and C. Block B has been overwritten several times. Instead of overwriting the old block B in the segment buffer, the new data of block B is simply appended to the segment buffer. The old data of block B is still in the buffer, although it becomes inaccessible (hence garbage). The *append-only* policy ensures that the ordering relationship between requests is maintained, and correctly recorded in the cache-disk. If the new copy of block B were to overwrite the old copy of the block B in the segment buffer, then order of updates between B and C would not have been preserved.

Although the append-only policy is necessary, it also has some drawbacks. Specifically, it increases the logging and destaging overhead since much garbage is logged into the cache-disk. One possible solution is to "compact" the segment buffer prior to writing it to the cache-disk. However the compacting process is a CPU-intensive task since it involves moving large amount of data in RAM. We have not tried this in our current implementation.

3.5.2 Reads

In DCD, data may be in one of three different places: the RAM buffer, the cache-disk and the data-disk. Conceptually read operations are simple; the driver just needs to find the locations of the requested data, get the data and return them to the host.

However, when we actually started the design and implementation, we found that reads are quite complex. Because of fragmentation, the contents of a requested data block may be scattered in all of the three places. The driver has to collect all data from all these places and assemble them together. This is not only complicated, but time-consuming.

Similar to writes, when a read request is received, the DCD driver searches the Data Lookup Table for entries. Most requests can not be found in the table, because newly written data is also cached in the file system. Therefore the system does not need to reread them. On the other hand the older written data has very likely been moved to the data-disk by destaging. Hu and Yang found in [1], that 99% of all read must be sent to the data disk. The simulations and experimental results confirm this; reads from other places seldom occur. Since virtually all reads are sent to the data-disk which has the same on-disk data layout as the traditional disk, DCD will at least have similar performance compared to a traditional disk [2].

For the majority of read requests, the DCD driver simply forwards them to the data-disk. For the remaining 1% of requests, if all the data are stored in the RAM buffer or consecutively on the cache-disk, the operation is also simple. However, for a

[2] In fact, compared to a traditional disk, DCD has better read performance because it removes the write traffic from the critical path therefore reducing the disk contention.

very small percentage of requests that access data scattered all over the system, the operation could be quite complex. After evaluating several alternatives, we adopted a simple solution: The driver first calls the *foreground destage* routine to move all the data related to the requests to the data-disk, then forwards the requests to the data-disk to fetch the data. While this approach may not be the most efficient one, it greatly simplifies our implementation. In addition, since such cases occur infrequently, the benefits of simple implementation override the low efficiency, and performance is not hurt noticeably.

3.5.3 Destages

In DCD, data written into the cache-disk must eventually be moved into the original location on the data-disk. This process is called *destaging*. The destage process starts when the DCD driver detects an idle period, a span of time during which there is no request. The threshold of the idle period (any idle period longer than the threshold will start the destage process) is a function of the cache-disk utilization. If the cache-disk is mostly empty, then the threshold can be as long as one second. However, if the cache-disk is mostly full, the threshold can be as short as 1 ms, to quickly empty the cache-disk, and prepare for the next burst of requests. Once the destage process starts, it will continue until the cache-disk becomes empty, or until a new request arrives. In the later case the destaging is suspended, until the driver detects another idle period.

The driver uses a "last-write-first-destage algorithm" [1], where the most recently written segment (the first one on the dirty segment list) is destaged first. While this algorithm is very straightforward, and works well, it may result in unnecessary destaging work since it always destages newly written data first. An optimization to this algorithm would be to use the segment-header's *valid-slot-count* to select a segment, for destage, which contains the least amount of valid data. Therefore reducing the number of writes needed to move the data from cache-disk to the data-disk. Also, when writing data blocks from the destage buffer to the data-disk, the driver does not try to reorder the writes. A possible optimization here is to reorder the blocks according their locations on the data disk to minimize the seek overhead. We will try these optimizations in the future version of the driver.

3.5.4 Crash Recovery

The crash recovery process of a system using DCD can be divided into two stages. In the first stage the DCD tries to recover itself to a stable state. In the second stage the file system performs normal crash-recovery activities.

The crash recovery of the DCD driver is relatively simple. Since cached-data is saved reliably to the cache-disk, only the in-memory data-structures, such as the *Data Lookup Table* and the *In-Memory Segment-Headers*, need to be reconstructed. We have designed the crash-recovery algorithm, but since it is not part of the critical path of the driver, this has not been been implemented in the current prototype.

To rebuild the in-memory data structures, the DCD driver can first read all of the *On-Disk Segment-Headers* from the cache-disk into RAM. For a cache-disk of 64 MB, only 512 headers need to be read. Since the cache-disk locations of the headers are known, this should not take too long. The headers in RAM are organized in a list ordered ascendingly according to their time-stamps. Starting with the first header on the list, the headers can be used in temporal order, to reconstruct the *Data Lookup Table* and the *In-Memory Segment-Headers*. Each time a new header is encountered, entries in the *Data Lookup Table* and *In-Memory Segment-Headers* may be overwritten, because data in the second segment may overwrite data in the earlier segment. This process continues until all the headers are processed. Essentially, all the cache-disk headers can be used to "replay" the entire history of requests captured on the cache-disk. At this point, the DCD driver returns to a clean and stable state, and the file system can start a normal crash-recovery process, such as running the "fsck" program.

3.6 Reliability and File System Integrity

DCD uses immediate report mode for most writes, except for those requests with the "sync" flags set by user applications. Write requests are returned from the driver once the data are written into the RAM buffer. This scheme does not cause a reliability problem for the following two reasons.

First, the delay caused by the DCD driver is limited to several hundreds of milliseconds at most. Once the previous log writing finishes (which may take up to several hundreds of milliseconds), the driver will write the current segment buffer to the cache-disk even before the segment buffer becomes full. Therefore the DCD driver guarantees that data be written in to the cache-disk within several hundreds of milliseconds. Since most file data is cached in RAM for 30–60 seconds before they are flushed into the disk anyway, we believe that the additional several hundreds of milliseconds should not result in any problem.

Second, for metadata updates, the "append-only" algorithm of the DCD driver guarantees the order of updates to the disk. So while metadata updates may be kept in RAM for up to several hundred milliseconds, they are guaranteed to be written into the disk in the proper order to ensure file system integrity. In this sense DCD is similar to other solutions such as LFS and Soft Updates, which also store metadata updates in RAM for even longer time but are able to maintain file system integrity.

3.7 Layered Device Driver Approach

One potential difficulty of the DCD device driver is that it may have to communicate with disk hardware directly, resulting in high complexity and poor portability. We solved this problem by using a *multi-layered device driver* approach. We implemented our DCD driver on top of a traditional disk device driver (End users specify which traditional disk driver will be used). The DCD driver calls the low-level disk driver, through the standard device driver interface, to perform the actual disk I/Os. This approach has three major advantages. First, it greatly simplifies the implementation efforts since the DCD driver avoids the complex task of direct communication with the hardware [3]. Second, the same DCD driver works with all kinds of disks in a system because all the low-level disk device drivers use the same standard interface. Third, it is easy to port our current implementation to other systems, since most Unix systems use similar device driver interfaces.

4 Performance Results

4.1 Benchmarks and Experimental Setup

To examine the performance of the experimental systems, we need some benchmark programs. Initially we have planed to use the Andrew benchmark [13]. However we found that the execution time of Andrew benchmark is very short (less than several seconds for most phases in the benchmark) on most modern machines, resulting in significant measurement errors. Also the benchmark does not exercise the disk system sufficiently in machines with reasonable amount of RAM, since most of its data can be cached in RAM.

Because of this, we have created a set of our own metadata intensive benchmark programs. Our benchmarks are similar to the ones used by Ganger and Patt in [3]. The benchmarks consist of 3 sets of programs: *N-user untar*, *N-user copy*, and *N-user remove*. *N-user untar* simulates one or more users concurrently and independently *un-taring* a tar file to create a directory tree. *N-user copy* and *N-user remove* simulate one or more users concurrently copying (using the "cp -r" command) or removing (using the "rm -r" command) the entire directory tree created by the *N-user untar* benchmark, respectively. We chose these benchmarks because they represent the typical and realistic I/O intensive tasks performed by normal users on a daily basis.

The particular tar file we used is the source-code distribution tar file of Apache 1.3.3. Apache is the most popular Web server as of this writing. The resulting source file tree contains 31 subdirectories and 508 files with an average file size of 9.9 KB. The total size of the directory tree is 4.9 MB.

To make our evaluation more conservative, we intentionally flush the contents of the kernel buffer cache by first unmounting then remounting the file system between each run. This creates a *cold read cache effect*, forcing the system to read data from the disk again for each benchmark run. Our measured results show that the performance improvement of the DCD driver over the traditional disk driver is much higher (up to 50% higher) when the kernel buffer cache is warm (that is, if we do not flush the contents of the buffer). The cold cache effect results in a large number of read requests that can not be improved significantly by DCD. Because of this artificially introduced cold cache effect, the performance results reported in the section should be viewed as a conservative estimate of DCD performance. Real world DCD performance should be even better, because the buffer cache will be warm most of the time.

All experiments were conducted on a Sun Ultra-1 Model 170 workstation running Solaris 2.6. The machine has 128 MB RAM and a 4 GB, high performance Seagate Barracuda hard disk drive. The system was configured as a Logical DCD with the first 64 MB of the disk used as the cache-disk partition of the DCD; the rest as the data-disk. The RAM buffer size of the DCD driver was 256 KB. The total RAM overhead, including the RAM buffer, the Data Lookup Table, the In-Memory Segment-Headers, etc., is about 750 KB, which is a small portion of the 128 MB system RAM.

The performance of the DCD device driver is com-

[3]Our prototype implementation consists of only 3200 lines of C code. Once we have validated our algorithms with an event-driven simulator, the first author took only two weeks to produce a working device driver.

pared to that of the default disk device driver of Solaris. We used the Unix *timex* command to measure the benchmark execution time, and the user and system CPU times. We also used the built-in kernel tracing facilities of Solaris to obtain detailed low-level performance numbers such as the average response times of read and write requests.

4.2 Measured Performance Results

In this subsection we compare the performance of our DCD device driver to that of the traditional disk device driver of Solaris. All numerical results shown here are the averages of three experimental runs.

4.2.1 Benchmark Execution Times

Figures 7–9 compare the benchmark execution times for the traditional disk device drivers with those of the DCD device driver. The total execution times are further broken down into the *CPU time* and the *waiting time*. The figures clearly show that these benchmarks run dramatically faster using the DCD driver than using the traditional driver. The DCD driver outperforms the traditional driver by a factor of 2–6 in all cases. The most impressive improvements come from the *N-user untar* and the *N-user remove* benchmarks that are very metadata intensive, where DCD runs more than 3–6 times faster than the traditional driver. The *N-user copy* benchmark spends much of its time on disk reads which can not be improved by DCD. Its performance improvement of 2–3 times is not that dramatic but still very impressive. Note that the performance difference is even bigger if we do not artificially flush the cache contents between each run. For example, our results show that the *N-user copy* benchmark also runs 4–5 times faster on the DCD driver than on the traditional driver if we do not flush the cache.

Figures 7–9 clearly indicate that the performance of the traditional driver is severely limited by the disk speed. Its CPU utilization ratio is only about 3–15%, meaning the system spends 85–97% of its time waiting for disk I/Os. While the performance of DCD is still limited by the disk speed, DCD significantly improves the CPU utilization ratio to 16–50%.

We can also see that the DCD driver and the traditional disk driver have similar CPU overheads. This demonstrates that our DCD driver algorithms are very efficient. Although the DCD driver is much more complex than the traditional disk driver, the former increases the CPU overhead only slightly compared to the later.

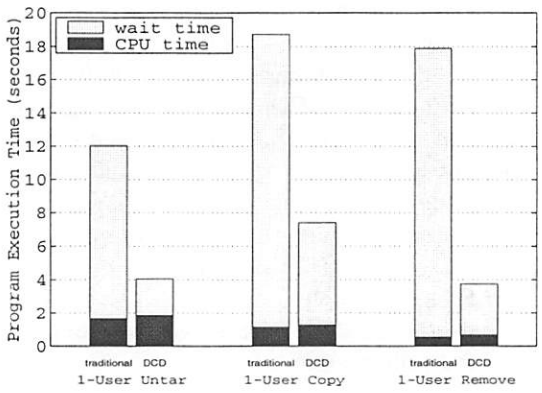


Figure 7: Benchmark execution times for *1-user untar, copy* and *remove*

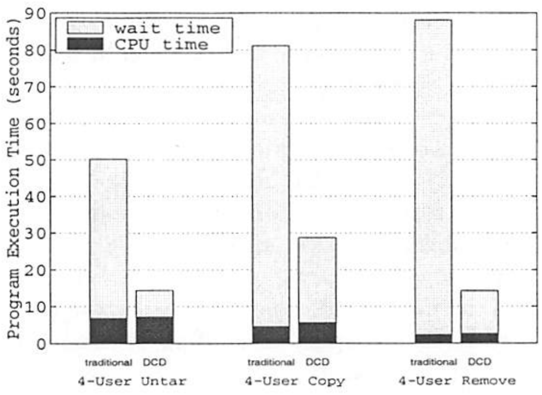


Figure 8: Benchmark execution times for *4-user untar, copy* and *remove*

4.2.2 Maximum Possible Speedup

We have shown that DCD can speed up the metadata intensive benchmarks by a factor of 2–6. To see if we can further improve the performance of DCD and to understand what is the performance limitation, we have measured the raw disk write bandwidth at various request sizes. The bandwidth is obtained by randomly writing 20 MB worth of data to the disk at different sizes, using the raw device driver interface.

Figure 10 shows the measurement results in terms of I/O bandwidth. The raw disk write performance at the 128 KB unit size is about 55 times faster than the 1 KB unit, 28 times faster than the 2 KB unit, 14.3 times faster than the 4 KB unit, and 7.4 times faster than the 8 KB unit sizes.

DCD always writes to the cache disk in the 128 KB segments. When running the *4-user remove* benchmark program on the traditional disk device driver, our profiling results show that about 50%

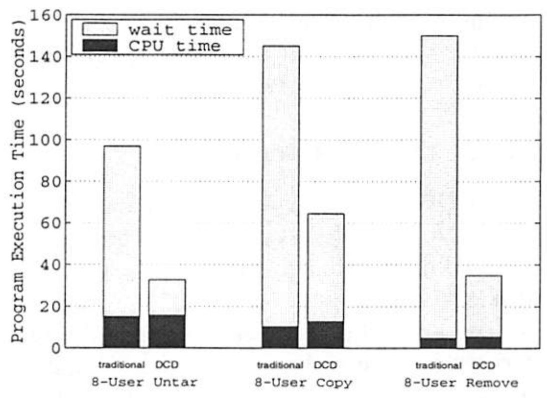


Figure 9: Benchmark execution times for *8-user untar, copy* and *remove*

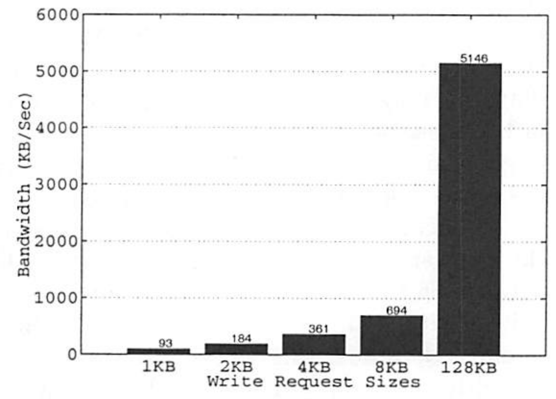


Figure 10: Raw disk write performance of the experimental system

of requests are in 8 KB, 15% in 2 KB and 35% in 1 KB. Therefore the maximum possible speedup of DCD over the traditional disk driver is approximately $7.4 * 50\% + 28 * 15\% + 55 * 35\% = 25.4$. The benchmarks also spend some time on reads and CPU overhead which can not be improved by DCD. For our *4-user remove* benchmark, the reads and CPU overhead account for about 7% of the total execution time on the traditional disk driver. As a result, we expect that the maximum achievable speedup be around $1/(7\% + 93\%/25.4) = 9.4$ without destage.

Our prototype DCD driver achieves a performance improvement of 6.2 times over the traditional device driver for the *4-user remove* benchmark. In other words, the prototype driver realizes two thirds of the potential performance gain. Although a small portion of this loss of gain is due to the overhead of destaging, our analysis indicates that there is clearly still room for further improvement in the implementation. One of the main performance limitations in the current implementation is our locking algorithm. Because of the time limitations, we used a very simple but ineffective locking scheme in the DCD driver for concurrency control. When a request arrives, we exclusively lock the entire cache for the request until the request is complete. This scheme significantly limits parallelism and has a very negative impact on the performance, especially for the multi-user environment. We plan to reduce the lock granularity to increase parallelism in the next version.

4.2.3 I/O Response Times

In order to further understand why the DCD device driver provides such superior performance compared to the traditional disk device driver, we measured the read and write response times of both the DCD and the traditional disk device driver, using the kernel tracing facilities of Solaris.

Figures 11 and 12 show the *histograms* of read and write response times of the DCD driver and the traditional driver. The data was collected while running the *4-user copy* benchmark. In these figures the X-axes are for response times and the Y-axes are for the percentage of requests. Note that the X-axes and Y-axes of the DCD driver and those of the traditional driver have a different scale. The figures for the traditional driver have an X-scale of 0–500 ms, while the figures of the DCD driver have an X-scale of 0–25 ms. Because the DCD driver has much lower response times, the histograms would have be crammed in the left part of the figures and become inscrutable had we used the same X-scale for all the figures.

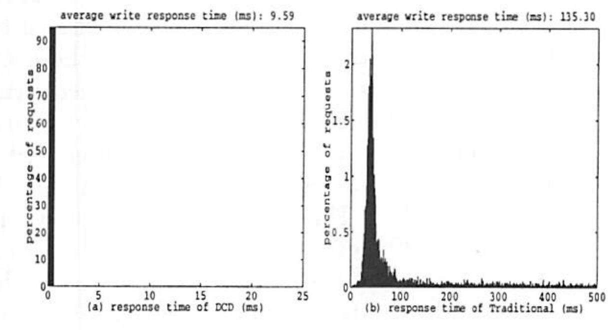


Figure 11: Histograms of Write Response Times of the DCD driver and Traditional Driver. *Virtually all requests here are synchronous.*

As shown in Figure 11, for the traditional disk driver, the response times for the majority of writes fall within the range of 25–150 ms. The average write

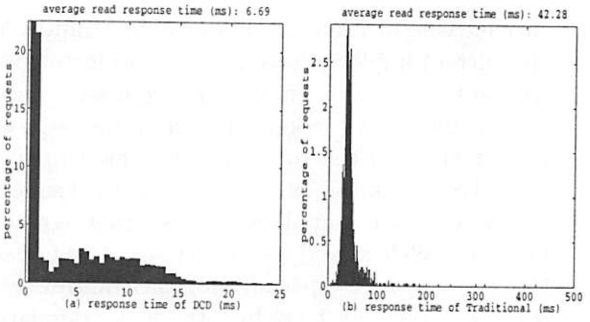


Figure 12: Histograms of Read Response Times of the DCD driver and Traditional Driver

response time is 135.30 ms. Such a long response time can mainly be attributed to the long queuing time caused by disk bandwidth contention among 4 processes.

The DCD driver, on the other hand, significantly reduces the response time of writes. About 95% of writes complete within 1ms, because they complete after being written into the RAM buffer. Another 3% of the writes, barely visible in Figure 11(a), take less than 25 ms. Finally, the remaining 2% of request takes about 25–1500 ms to finish. This happens when the cache-disk can not finish log writing quickly so the incoming requests have to wait. The average write response time of the DCD driver is 9.59 ms [4], which is is less than 1/14 of the response time of the traditional disk driver.

While the DCD driver can not speed up the read requests directly, it significantly reduces the average read response time in two ways. First, since DCD reduces write traffic at the data-disk, the data-disk has more available bandwidth for processing read requests. This eliminates the queuing time caused by disk bandwidth contention and decreases the average read response time. Second, the reduced write traffic at the data-disk allows a more effective use of the built-in hard disk cache. Since fewer writes 'bump' read data, the read-ahead cache hit ratio is significantly higher. This is shown in Figure 12 where approximately 40% of read requests complete within 1 ms because of the read-ahead cache hits. On the other hand, in the traditional driver very few read requests finish within 1 ms. This is because a large number of small writes constantly flush the read-ahead cache, rendering the cache virtually useless. Both the reduced bandwidth contention and the improved cache hit-ratio "shifts" the entire histogram to the far left. The average read response time of DCD is 6.69 ms, which is about 1/7 of the 42.28 ms response time of the traditional disk driver.

[4]This number is surprisingly high, considering the fact that 95% of requests finishes within 1ms. The bias is caused by the 2% of requests that finish within 25–1500 ms.

4.2.4 Destage Overhead

While DCD shows superb performance, for each data write, DCD has to do extra work compared to a traditional disk. It has to write the data into the cache-disk first and read it back from the cache-disk later for destage. This extra overhead may become a performance concern. However, we found that this extra work does not result in extra traffic in the disk system.

In DCD, all writes to and reads from the cache-disk are performed in large segment sizes — typically up to 32 requests can be written into or read from the cache-disk, both in one disk access. Therefore the overhead increases only slightly. Moreover, this small overhead can be compensated by the fact that data can stay in the cache-disk much longer than in RAM because the cache-disk is more reliable and larger than the RAM cache. As a result, the data in the cache-disk has a better chance of being overwritten, reducing the number of destaging operations. Hartman and Ousterhout demonstrated in [14] that between 36%–63% of newly written bytes are overwritten within 30 seconds and 60%–95% within 1000 seconds. Because of this, when compared to a normal disk driver, a DCD driver actually generates *fewer* total disk requests (including requests from the system and applications as well as the internal destaging requests).

5 Conclusion

In this paper we have designed and implemented a Disk Caching Disk (DCD) device driver for the Solaris operating system. Measured performance results are very promising, confirming the simulation results in [1, 2]. For metadata intensive benchmarks, our un-optimized DCD driver prototype outperforms the traditional system by a factor of 2–6 in terms of program execution speeds. The driver also guarantees file system integrity in the events of system crashes or failures. Moreover, unlike previous approaches, the DCD driver is completely transparent to the OS. It does not require any changes to the OS or the on-disk data layout. As a result, it can be used as a "drop-in" replacement for the traditional disk device driver in an existing system to obtain immediate performance improvement.

We plan to make the source code and binary code of our DCD device driver available to the public as soon as possible. We also plan to port the driver to other systems such as FreeBSD or Linux.

Acknowledgments

The authors would like to thank the anonymous referees for their advice on improving this paper, and Jordan Hubbard and Terry Lambert for providing information about the status of LFS and Soft Updates in FreeBSD.

References

[1] Y. Hu and Q. Yang, "DCD—disk caching disk: A new approach for boosting I/O performance," in *Proceedings of the 23rd International Symposium on Computer Architecture*, (Philadelphia, Pennsylvania), pp. 169–178, May 1996.

[2] Y. Hu and Q. Yang, "A new hierarchical disk architecture," *IEEE Micro*, vol. 18, pp. 64–76, November/December 1998.

[3] G. R. Ganger and Y. N. Patt, "Metadata update performance in file systems," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 49–60, Nov. 1994.

[4] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, "An implementation of a log-structured file system for UNIX," in *Proceedings of Winter 1993 USENIX*, (San Diego, CA), pp. 307–326, Jan. 1993.

[5] J. Ousterhout and F. Douglis, "Beating the I/O bottleneck: A case for log-structured file systems," tech. rep., Computer Science Division, Electrical Engineering and Computer Sciences, University of California at Berkeley, Oct. 1988.

[6] M. Rosenblum and J. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, pp. 26 – 52, Feb. 1992.

[7] U. Vahalia, *UNIX Internals — The New Frontiers*. Prentice Hall, 1996.

[8] Q. Yang and Y. Hu, "System for destaging data during idle time by transferring to destage buffer, marking segment blank, reordering data in buffer, and transferring to beginning of segment." U.S. Patent No. 5,754,888, May 1998.

[9] K. Elhardt and R. Bayer, "A database cache for high performance and fast restart in database systems," *ACM Transactions on Database Systems*, vol. 9, pp. 503–525, Dec. 1984.

[10] T. Blackwell, J. Harris, and M. Seltzer, "Heuristic cleaning algorithms in log-structured file systems," in *Proceedings of the 1995 USENIX Technical Conference: January 16–20, 1995, New Orleans, Louisiana, USA* (USENIX Association, ed.), (Berkeley, CA, USA), pp. 277–288, USENIX, Jan. 1995.

[11] J. C. Mogul, "A better update policy," in *Proceedings of the Summer 1994 USENIX Conference* (USENIX Association, ed.), (Berkeley, CA, USA), pp. 99–111, USENIX, June 1994.

[12] G. R. Ganger and Y. N. Patt, "Using system-level models to evaluate I/O subsystem designs," *IEEE Transactions on Computers*, pp. 667–678, June 1998.

[13] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems*, vol. 6, pp. 51–81, Feb. 1988.

[14] J. H. Hartman and J. K. Ousterhout, "letter to the editor," *Operating Systems Review*, vol. 27, no. 1, pp. 7–9, 1993.

An Application-Aware Data Storage Model *

Todd A. Anderson, James Griffioen
Department of Computer Science
University of Kentucky

Abstract

We describe a new application-controlled file persistence model in which applications select the desired stability from a range of persistence guarantees. This new abstraction extends conventional abstractions by allowing applications to specify a file's volatility and methods for automatic reconstruction in case of loss. The model allows applications, particularly ones with weak persistence requirements, to leverage the memory space of other machines to improve their performance. An automated (filename-matching) interface permits legacy applications to take advantage of the variable persistence guarantees without being modified. Our prototype implementation shows significant speed-ups, in some cases more than an order of magnitude over conventional network file systems such as NFS version 3.

1 Introduction

In recent years, workstations and personal computers have become extremely powerful and have seen impressive increases in storage capacities. The cost effectiveness of these systems combined with emerging high-speed network technologies have led to local and intra-area networks of workstations with tens of gigabytes of memory storage and hundreds of gigabytes of disk storage. Studies show that the aggregate capacity and processing power of these systems are grossly underutilized. Acharya and others [1] discovered that, on average, 60% to 80% of a network's workstations are idle at any point and 20% to 70% of workstations are always available. Once a workstation has been idle 5 minutes, it will remain idle for an average of 40 to 90 minutes, implying that idle machines can be selected with high confidence that they will remain idle.

While aggregate processing power has grown sharply, applications have seen only modest performance improvements [21]. This is due in large part to the historic reliance of file systems on disk-based storage, characterized by slow access times. Moreover, the performance gap between disks and memory/remote memory is increasing. Several file system designs have noted the growing gap and have proposed ways to improve file system performance. Local client caching is a well-known technique to improve read times. More recent cooperative caching systems [10, 13, 23] have extended the caching model to use memory of other workstations in the system to improve the cache hit rate. Systems such as xFS [3] have improved write performance by striping data across the aggregate disk capacity of the system. Other approaches are described in Section 8.

Our goal is to shrink the gap between file system performance and rapidly improving hardware performance by introducing a new file system abstraction that can capitalize on the large aggregate idle resources of current distributed systems. We extend past approaches by allowing the application to control the tradeoff between persistence and performance.

2 Derby

In our previous work, we investigated a new file system design called Derby [15, 14] that used idle remote memory for both read and write traffic. Reads and writes occurred at remote memory speeds while providing the disk persistence necessary for database transaction-oriented storage. Derby assumes a small fraction of the workstations were equipped with uninterruptable power supplies (Workstations with UPS or *WUPS*). The system operates as follows. All active data resides in memory. Read requests that cannot be satisfied from the local cache use a dynamic address table lookup to find the idle machine that holds the data. The request is sent to a server process on the remote machine that returns the data from its memory. Write requests occur in a similar

*This work supported in part by NSF grant numbers CCR-9309176, CDA-9320179, CDA-9502645 and DARPA grant number DAAH04-96-1-0327.

manner but also send the written/modified data to one or more WUPS machines. The data is held temporarily in WUPS memory until the server process asynchronously writes the data to disk and informs the WUPS that the newly written data can be removed. By using WUPS for short-term persistence and disks for long-term persistence, Derby achieves disk persistence at remote memory speeds.

The advantage of Derby and similar main memory storage systems [18, 17, 24] is the ability to achieve traditional disk persistence at memory-speeds. However, disk persistence comes at a price. Such systems require special purpose hardware such as NVRAM or uninterruptable power supplies. In Derby, disk persistence increases the communication overhead between clients, servers, and WUPS servers. In addition, the disk system poses a potential bottleneck because *all* write traffic (including small, frequent operations such as file creation, deletion, and other metadata changes) hits the disk. Note that past file system analysis shows average file lifetimes to be quite short (80% of files and 50% of bytes last less than 10 minutes [4]). Thus, many files are likely to be written to cache and disk, read from the cache, and deleted, without ever being read directly from disk. This unnecessarily consumes CPU, bus, memory, disk bandwidth, and network resources in a distributed file system.

3 Enhancing Derby

Given the overhead and performance drawbacks associated with disk persistence, a file system that offers multiple persistence guarantees rather than a "one-size-fits-all model" has several potential benefits. Consider the Unix Temporary File System (tmpfs) which stores data in the high-speed (volatile) memory of the local machine and never writes the data to disk. Despite the potential for data loss, many applications are willing to take that chance in exchange for fast file access. Unfortunately, to use tmpfs, users must know where the tmpfs file system is mounted (assuming it is mounted) and must selectively place their temporary files in the tmpfs portion of the directory structure. This also forces logically related files with different persistence requirements to be stored in different places. As another example, local file systems often delay writes to disk to improve performance. In this case, all data eventually hits the disk, but recently written data may be lost, often to the surprise and dismay of the user. We conclude that applications are often willing to trade persistence for performance and that a one-size-fits-all persistence model will be suboptimal for most applications. In this paper, we develop a new file system that supports a *per-file, selectable-persistence* model.

Our analysis shows that the majority of data written to the file system can tolerate a persistence level between tmpfs and disk persistence. Note, this does not mean "the data will be lost" but rather that we will accept a slightly higher probability of loss in exchange for better performance. For example, web-browser cache files can be lost without ill effect. Locally generated object files (.o's) and output files from word processing and typesetting applications can be easily regenerated or replaced. Likewise, whether adding to or extracting from an archive file (for example, tar or zip), the resultant file(s) can be recreated as long as the original file(s) survive. Even files initially retrieved by HTTP or FTP, if lost, usually can be re-downloaded. Certainly, there are exceptions to the above generalizations and so a per-file persistence model is necessary to provide the correct persistence for atypical files.

To quantify the amount of file data that can take advantage of a selectable persistence abstraction, we snooped NFS traffic on our computer science networks. The trace was conducted for 3 days and recorded all NFS traffic sent to our 5 file server machines running Solaris 2.6. All user accounts reside on the file servers as is typical of many distributed file systems. The majority of activity consisted of reading mail, browsing the web (browser cache files), editing/word-processing/typesetting, compiling, and archiving (for example, tar and gzip). We recorded the names of all files opened for writing and the number of bytes written to each file. We then classified files as requiring disk persistence or weak persistence (something less than disk persistence). If there was any doubt as to the persistence required by a file, we erred on the side of caution and classified the file as requiring disk persistence stability. Note that the amount of weakly persistent data would be higher had traffic to `/tmp` been captured as part of this trace. The results of the study are reported in Table 1.

File Type	**# of files**	**# of bytes**
Weak persistence	24477 (75%)	4,410,711,305 (97%)
Disk persistence	8165 (25%)	116,717,201 (3%)

Table 1: Number of files/bytes written or modified.

Although 25% of files required disk persistence, most of these files were small text files (.h and .c files for example) created with an editor and did not require memory-speed write performance. Disk-persistent files also only accounted for a small portion of the total file system write traffic. Conversely, weakly persistent files made up the bulk of write traffic, consisting of compiler and linker output, LaTeX output (for example, .dvi, .log, and .aux files), tar files, Netscape cache files, and temporary files created by a variety of programs. From this we conclude that a large percentage of write traffic would trade weaker persistence guarantees for performance. Furthermore, we believe that the aggregate memory space of current networks is ideal for storing weakly-persistent data requiring fast access.

4 MBFS Overview

Our objective was to replace Derby's (and conventional file systems') one-size-fits-all persistence model with a configurable persistence model thereby removing the need for non-volatile memory and the other overheads of disk persistence. The new design, called *MBFS (Memory Based File System)*, is motivated by the fact that current distributed systems contain massive amounts of memory storage - in many cases tens or hundreds of gigabytes. The aggregate memory space, consisting of both local and remote memory, can be used to provide high-speed file storage for a large portion of file traffic that does not require conventional disk persistence. These files are often short lived and require fast access. In MBFS, memory rather than disk serves as the primary storage location for these files. Disks, tapes, writable CD's, and other high-latency devices are relegated to the role of archival storage, out of the critical path of most file operations.

MBFS supports a continuum of persistence guarantees and allows applications to select the desired persistence on a per-file basis. Weaker persistence guarantees result in better performance than strong persistence guarantees, allowing applications with weak persistence requirements to avoid the performance penalties and overheads of conventional disk-persistent models.

Because systems cannot guarantee persistence (data will not be lost under any circumstance), they actually only guarantee "the probability of persistence" or "the probability the data will not be lost." In conventional file systems, persistence is defined as (1-(probability of disk failure)) , whereas MBFS supports any persistence probability. The difficulty in supporting such a continuum lies in an application's need to know exactly what each position in the continuum means. For example, what does it mean to select a persistence at the midpoint of the continuum, halfway between "will lose the data immediately" and "will keep multiple almost indestructible copies"? The midpoint definition depends on the the devices used to store the data, their failure characteristics, and if or when data is archived to more persistent storage. Also, the same persistence probability can be implemented different ways. For example, storing data in two NVRAM's may have an equivalent persistence probability to storing the data on an inexpensive disk. Exposing applications to the details of the storage devices and their failure rates is undesirable as it complicates the file system interface and ties the application to the environment, thereby limiting application portability. As a result, MBFS attempts to hide these details from the application.

4.1 Storage Hierarchy

To clarify the "guarantees" provided at different settings of the persistence spectrum without binding the application to a specific environment or set of storage devices, MBFS implements the continuum, in part, with a *logical storage hierarchy*. The hierarchy is defined by N levels:

1. **LM (Local Memory storage):** very high-speed volatile storage located on the machine creating the file.
2. **LCM (Loosely Coupled Memory storage):** high-speed volatile storage consisting of the idle memory space available across the system.
3. **-N DA (Distributed Archival storage):** slower speed stable storage space located across the system.

Logically, decreasing levels of the hierarchy are characterized by stronger persistence, larger storage capacity, and slower access times. The LM level is simply locally addressable memory (whether on or off CPU). The LCM level combines the idle memory of machines throughout the system into a loosely coupled, and constantly changing, storage space. The DA level may actually consist of any number of sublevels (denoted $DA_1, DA_2, ..., DA_n$) each of increasing persistence (or capacity) and decreasing performance. LM data will be lost if the current machine crashes or loses power. LCM data has the potential to be lost if one or more machines crash or lose power. DA data is guaranteed to survive power outages and machine crashes. Replication and error correction are provided at the LCM and DA levels to improve the persistence offered by those levels.

Each level of the logical MBFS hierarchy is ultimately implemented by a physical storage device. LM is implemented using standard RAM on the local machine and LCM using the idle memory of workstations throughout the network. The DA sub-levels must be mapped to some organization of the available archival storage devices in the system. The system administrator is expected to define the mapping via a system configuration file. For example, DA-1 might be mapped to the distributed disk system while DA-2 is mapped to the distributed tape system.

Because applications are written using the logical hierarchy, they can be run in any environment, regardless of the mapping. The persistence guarantees provided by the three main levels of the hierarchy (LM, LCM, DA_1) are well defined. In general, applications can use the other layers of the DA to achieve higher persistence guarantees, without knowing the exact details of the persistence guaranteed; only that it is better. For applications that want to change their storage behavior based on the characteristics of the current environment, the details of each DA's persistence guarantees, such as the expected mean-time-till-failure, can be obtained via a **stat()** call to the file system. Thus, MBFS makes the layering abstraction explicit while hiding the details of the devices used to implement it. Applications can control persistence with or without exact knowledge of the characteristics of the hardware used to implement it.

Once the desired persistence level has been selected, MBFS's loosely coupled memory system uses an addressing algorithm to distribute data to idle machines and employs a migration algorithm to move data off machines that change from idle to active. The details of the addressing and migration algorithms can be found in [15, 14] and are also used by the archival storage levels. Finally, MBFS provides whole-file consistency via callbacks similar to Andrew[19] and a Unix security and protection model.

4.2 File Persistence

The continuity of the persistence spectrum is provided by a set of per-file *time constraints* (one for each level of the storage hierarchy) that specify the *maximum* amount of time that modified data may reside at each level before being archived to the next level of the storage hierarchy. Since each level of the storage hierarchy has increasing persistence guarantees, the sooner data is archived to the next level, the lower its possibility of loss. Conversely, if data plans to linger in the higher levels, the chance of that data surviving decreases. A daemon process tracks modified data at each storage level, archiving the data to the next lower level within the prescribed amount of time. When data is archived to the next level, it continues to exist at the current level unless space becomes scarce.

Conceptually, as the cost of recreating the data increases, so should the file's persistence level. For example, while a file may be relatively easy to regenerate (for example, an executable compiled from sources) and may never need to hit stable storage, moving it to stable storage after some period of time can prevent the need for the data to be regenerated later (for example, after a power failure). If a file is highly active (i.e., being written frequently), such as an object code file (.o), the time delay prevents the intermediate versions of the file from being sent to lower levels of the hierarchy but will ultimately archive the data at some point in the future, after the active period has finished.

Although archiving of data to lower levels of the hierarchy occurs at predefined time intervals, there are several other events that can cause the archival process to occur earlier than the maximum time duration. First, the daemon may send the data to the next level sooner than required if the archiving can be done without adversely affecting other processes in the system. Most networks of workstations experience an inactive period each night. Consequently, modified files are often archived through all the levels at the end of each day regardless of how long their respective time constraints are. Second, some storage levels such as LM and LCM are limited in capacity. At such levels, modified data may need to be sent to the next level early to make room for new data. Lastly, a modified file may be archived to stable storage (DA level) earlier than required if one of the file's dependencies is about to be changed (see Section 6).

5 MBFS Interface

In order to support variable persistence guarantees, we designed and implemented a new file system abstraction with two major extensions to conventional file system interfaces: a *storage abstraction extension* and a *reconstruction extension.* The storage abstraction extension allows programmers to specify the volatility of a file. MBFS stores the low-level volatility specification as part of the file's metadata and uses callbacks to maintain metadata and block consistency. The reconstruction extensions allows the system to automatically recreate most data that has been lost. The reconstruction extension is described in Section 6.

Although the MBFS logical storage hierarchy allows applications to specify the desired persistence, programmers will not use the system or will not make effective use of the system if the interface is too complex or difficult to select the correct persistence/performance tradeoff. While we expect many performance critical applications (for example, compilers, typesetters, web browsers, archival and compression tools, FTP) will be modified to optimize their performance using the new file system abstraction, we would like to offer MBFS's features to existing applications without the need to change them. In short, we would like to simplify the interface without sacrificing functionality. To that end, we use a multi-level design: a kernel-level raw interface that provides full control over the MBFS logical storage hierarchy and programmer-level interfaces that build on the raw level to provide an easy-to-use interface to applications. The following describes the raw *kernel-level interface* and two programmer-level interfaces, the *category interface* and the *filename matching interface.* Note that additional interface libraries (such as file mean-time-to-failure) could also be implemented on top of MBFS's raw interface. Each library is responsible for mapping their respective volatility specifications to the low-level storage details that the kernel interface expects. Although we describe our extensions as enhancements to the conventional file system abstraction, the extension could also be cleanly incorporated into emerging extensible file system standards such as the Scalable I/O Initiative's proposed PFS[8].

5.1 Kernel Interface

The kernel interface provides applications and advanced programmers with complete control over the lowest-level details of MBFS's persistence specification. Applications can select the desired storage level and the amount of time data may reside at a level before being archived. Applications must be rewritten to take advantage of this interface. For this interface, only the open() routines are modified to provide volatility specification and to support the reconstruction extension.[1] Consequently, a file's volatility specification can be changed *only* when a file is opened. The open() call introduces two new parameters: one for specifying the volatility and one for specifying the reconstruction extension. The reconstruction parameter is discussed in Section 6.1.

The volatility specification defines a file's persistence requirements at various points in its lifetime. The volatility specification consists of a *count* and a variable size array of *mbfs_storage_level structures* described below. Each entry in the array corresponds to a level in the logical storage hierarchy (LM, LCM, $DA_1, ..., DA_n$). A NULL volatility specification is replaced by a *default volatility specification* defined by a system or user configuration file. The default volatility specification is typically loaded into the environment by the login shell or read by a call to a C-level library.

```
typedef struct {
    struct timeval   time_till_next_level;
    void            *replication_type;
} mbfs_storage_level;
```

time_till_next_level: Specifies the maximum amount of time that newly written or modified data can reside at this level without being archived to the next level of the hierarchy. Values greater than or equal to 0 mean that write() operations to this level will return immediately and the newly written data will be archived to the next level within time_till_next_level time. Two special values of time_till_next_level can be used to block future write() and close() operations. UNTIL_CLOSE (-1) blocks the close operation until all file blocks reach the next level. BEFORE_WRITE (-2) blocks subsequent write() operations at this level until the data reaches the next level.

replication_type: This is used to specify the type of replication to use at this level.

For increased persistence, availability, and performance, MBFS supports data replication and striping as specified by the *replication_type* field. The replication_type field defines the type and degree of replication used for that level as defined by the following structure:

[1]If the system supports other file open or file creation system calls (for example, the Unix creat() call), these calls also need to be modified to include a volatility and reconstruction extension

```
typedef struct {
  int  Type;
  int  Degree;
} mbfs_replication;
```

Type: A literal corresponding to the desired form of replication selected from SINGLE_COPY, MIRRORING, and STRIPING [22]. SINGLE_COPY provides no replication. MIRRORING saves multiple copies on different machines. STRIPING distributes a single file and check bits across multiple machines. The default is SINGLE_COPY.

Degree: The number of machines to replicate the data on. If *Type* is SINGLE_COPY this field is ignored. If MIRRORING, it defines the number of copies. If STRIPING, it defines the size of the stripe group.

Mirroring and striping increase reliability by ensuring data persists across single machine or disk failures. Because unexpected machine failures are not uncommon in a distributed system (for example, the OS crashes, a user accidentally or intentionally reboots a machine, user accidentally unplugs machine), replication at the LCM level greatly increases the probability LCM data will survive these common failures.

The kernel-level interface also requires modifications to the system calls used to obtain a file's status or a file system's configuration information (for example, stat() and statvfs() in Solaris). For applications requiring complete knowledge of the environment, MBFS returns information about a file's persistence requirement, reconstruction information, or the estimated mean time to failure of each of the DA levels based on manufacturer's specifications. The raw kernel-level interface provides full control over a file's persistence, but this control comes at the price of elegance and requires that the application provide a substantial amount of detailed information on each open() call.

5.2 Category Interface

To simplify the task of specifying a persistence guarantee, the MBFS interface includes a user-level library called the *category interface.* The premise of the category interface is that many files have similar persistence requirements that can be classified into persistence categories. Thus, the category interface allows applications to select the category which most resembles the file to be created. Category names are predefined or user-defined ASCII character strings that are specified in the open() call. The open() call optionally also takes a reconstruction parameter like the kernel interface.

The category library maps category names to full volatility specifications and then invokes the raw kernel-level interface. The mapping is stored in a process' environment variables which are typically loaded at login from a system or user-specific category configuration file. The environment variable is a list of (category name, volatility specification) pairs.

The system configuration file must minimally define the categories listed below to ensure portability of applications. Programmers and applications may also create any number of additional custom categories. The (minimal) system categories are divided into two sets. The first set defines categories based on the class of applications that use the file. The second set defines categories that span the persistence continuum. The continuum categories are useful for files that do not obviously fall into any of the predefined application classes. The application categories are:

EDITED: Files that are manually created or manually edited and typically require strong persistence (for example, source code files, email messages, text documents, word processing documents).

GENERATED: Files generated as output from programs that require very weak persistence because they can be easily recreated (for example, object files, temporary or intermediate file formats such as *.aux, *.dvi, *.log, and executables generated from source code).

MULTIMEDIA: Video, audio, and image files that are down-loaded or copied (as opposed to edited or captured multimedia data) such as gif, jpeg, mpeg, or wav files.

COLLECTION: A collection of files (archive) create from other files in the system or down-load (for example, *.Z, *.gz, *.tar, *.zip)

DATABASE: Database files often of large size, requiring strong persistence, high-availability, and top performance.

Categories that span the persistence spectrum are (from most volatile to least volatile):

DISPOSABLE: Data that is to be immediately discarded (such as /dev/null).

TEMPORARY: Temporary files that can be discarded if necessary and which will not reach DA_1.

EVENTUAL: Data can be easily recreated but should reach DA_1 if it lives long enough (several hours or more).

SOMETIME: Reconstructable data that is likely to be modified or deleted soon. If the data is not modified soon, the data should be archived to DA_1 relatively soon to minimize the need for reconstruction (repeatedly generated object files).

SOON: Data that should be sent to DA_1 as soon as possible, but it is not a disaster if the data is lost within a few seconds of the write.

SAFE: Data is guaranteed to be written to DA_1 before the write operation completes.

ROBUST: Data is stored at two or more DA levels.

PARANOID: Data is replicated at multiple DA levels of the storage hierarchy.

5.3 Filename Matching Interface

The filename matching interface does not require any changes or extensions to the conventional `open()` operation in order to obtain variable persistence guarantees. Instead, it determines the volatility specification from the filename. The filename matching interface allows the user to define categories of files based on filenames. Consequently, applications (or existing software libraries) need not be rewritten to use the filename matching interface. To take advantage of MBFS's variable persistence, applications can either be re-linked or can install an `open` system call hook that intercepts all `open` operations and redirects them to the filename matching library. This allows legacy applications to benefit from MBFS's variable persistence model without being modified in any way.

The filename matching library maps filenames to full volatility specifications needed by the kernel. Like the category interface, the mapping is stored in environment variables loaded from system or user-defined configuration files. The mapping consists of (regular expression, volatility specification) pairs. At `open` time, the library matches the filename being opened to a regular expression in the environment. If successful, the library uses the associated volatility specification, otherwise a default persistence is assigned.

5.4 Metadata and Directories

In most file systems, write() operations modify both the file data and the file's metadata (for example, file size, modification time, access time). Although it is possible to allow separate persistence guarantees for a file's metadata and data, if either the metadata or the data is lost, the file becomes unusable. Moreover, separate volatility specifications only complicates the file system abstraction. Consequently, MBFS's volatility specifications apply to both the file's data and its metadata.

A similar problem arises is determining how the volatility for directories is specified. In MBFS, directory volatility definitions differ from file volatility in two important ways.

First, all modifications to directory information (*e.g.,* file create/delete/modify operations) must reach the LCM immediately so that all clients have a consistent view of directory information. Only the metadata needs to be sent to LCM. The file's modified data can stay in the machine's LM if the file's metadata was sent to the LCM and a callback is registered with the LCM, and the file's LM timeout has not occurred.

Second, a directory inherits its volatility specification from the most persistent file created, deleted, or modified in the directory since the last time the directory was archived. If a file is created with a volatility specification that is "more persistent" than the directory's current volatility specification, the directory's specification must be dynamically upgraded. If the directory was lost and the directory's persistence wasn't greater than or equal to its most persistent file, the file would be lost from the directory (even if the file's data is not lost). Once the directory is archived to level N, the directory's volatility specification for level N can be reset to infinity. This produces optimal directory performance. Assigning stronger persistence guarantees to directories than the files they contain degrades performance and wastes resources because of the unnecessary persistence costs.

6 Reconstruction

The volatile nature of MBFS may result in lost file data due to unexpected failures. To encourage programmers and applications to select weaker persistence guarantees, MBFS aids in the process of restoring lost data. It should be noted that restoration is a mechanism that is applicable to many, but not necessarily all, files. In other words, the restoration mechanism cannot guarantee successful reconstruction. Some aspects of the environment (for example, date, time, software versions, access to the network) in which the program was originally run may be impossible to capture or recreate and may render reconstruction impossible. In these few cases, manual reconstruction or disk persistence should be used. However, if data is committed to stable storage within some reasonably short period of time, such as 24 hours, the environment is unlikely to change significantly during the volatile interval when reconstruction may be necessary. Therefore, we believe a large percentage of performance-critical files can use the reconstruction mechanism to achieve improved performance without fear of data loss or manual restoration.

MBFS supports two methods for restoring lost data: *manual reconstruction* and *automatic reconstruction.* The most obvious and painful method is manual reconstruction in which users manually reruns the program that originally created the lost file. The second approach relieves the user of this burden by having the file system automatically reconstruct lost files after system failures. Prior to a failure, the user or application specifies the information needed to reconstruct the volatile file. Using this information, the file system automatically reconstructs the lost data by re-executing the appropriate program in a correct environment.

6.1 The Reconstruction Extension

The MBFS open() call in the kernel, category, and filename matching interfaces optionally support a reconstruction parameter defined by the following structure:

```
typedef struct {
    char *reconstruction_rule;
    envp *environment[];
    int  num_dependencies;
    char *dependencies[];
    int  data_freshness;
    Bool reconstruct_immediately;
} mbfs_reconstruction;
```

reconstruction_rule: A string containing the command-line that the system should execute to reconstruct the file.

environment: An array of environment variables and values to be set in the shell before invocation of this file's reconstruction rule. If NULL, the current environment in assumed.

data_freshness: Specifies how "up-to-date" the contents should be with respect to it's dependencies:

- LATEST_DATA - This corresponds to a guarantee that the file will always contain the newest possible data. The file should be reconstructed if any of its dependencies have changed. This feature can be used to implement intentional files.
- VERSION_DATA - The system guarantees that some version of the file data exists, not necessarily the latest. Unlike LATEST_DATA, the file's contents are not regenerated when a dependency changes. Only if the data is lost will the data be regenerated. If a dependency is deleted, the reconstruction rule cannot be executed, so the system immediately archives the data to an DA level to ensure its persistence.
- OLD_DATA - This guarantees that the file's contents will be based on versions of the dependencies as they existed at the time the file was created. Before the system will allow a dependency to be changed or deleted, the system will ensure the data reaches an DA level because reconstruction might produce different data.

reconstruct_immediately: A boolean specifying when the system should execute the file's reconstruction rule.

- TRUE - The system will invoke the reconstruction rule as soon as lost file data is detected or, if LATEST_DATA is selected, as soon as a dependency is changed.
- FALSE - The system may postpone the invocation of the reconstruction rule for any amount of time up until the file is referenced. This is the default.

num_dependencies: The number of entries in the dependency array.

dependencies: An array of entries corresponding to the MBFS files on which the given file is dependent.

With the inclusion of dependencies and the LATEST_DATA freshness quality, the reconstruction extension naturally provides an automatic form of the

Unix "make" utility and provides a form of intensional files. While an automatic "make" is useful in some circumstances, users may still invoke the standard make utility manually.

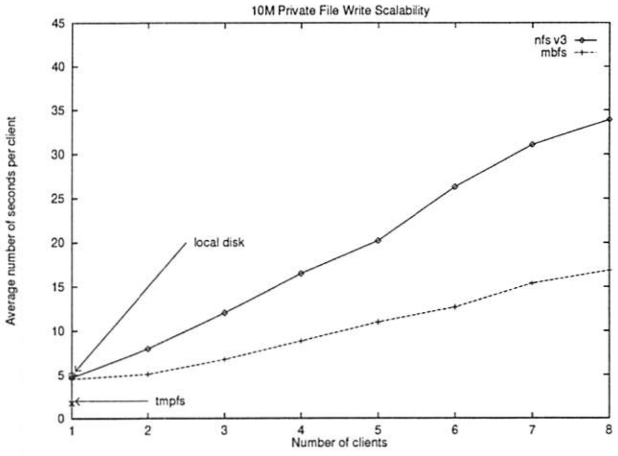
6.2 Reconstruction Environment

The system must determine on which machine the rule should be run. Remembering the machine that created the file is useful but not sufficient. First, the original machine may be down or heavily load at reconstruction time. Second, the original machine's environment may have changed since the file was created. Before a reconstruction rule is saved, MBFS collects the architecture and operating system of the machine and stores them with the reconstruction rule. At reconstruction time, the system searches for machines satisfying both architecture and OS, and chooses one of them on which to run the reconstruction rule. If no machine is found, an error message is sent to the owner of the file saying the file could not be reconstructed.

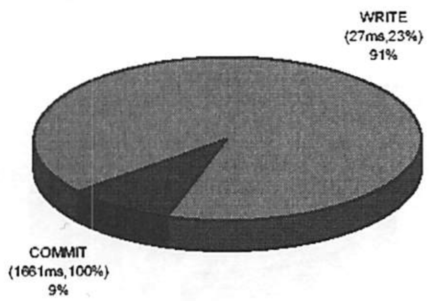
7 MBFS Prototype

To quantify the potential performance gains of a variable persistence model, we implemented a prototype MBFS system with support for LM, LCM, and DA_1 (via disks). The prototype runs on Solaris and Linux systems. The current prototype does not yet support the reconstruction extension. We then ran tests using five distinct workloads to evaluate which data can take advantage of variable persistence and to quantify the performance improvements that should be expected for each type of workload. Our tests compared MBFS against NFS version 3, UFS (local disk), and tmpfs. Only one point for UFS and tmpfs per graph are provided since distributed scalability is not an issue associated with localized file systems. In all but the edit test, no file data required disk persistence and typical performance improvements were in the range of three to seven times faster than NFS v3 with some tests almost two orders of magnitude faster.

The MBFS server runs as a multi-threaded user-level Unix process, experiencing standard user-level overheads (as opposed to the NFS server which is in-kernel). Putting the MBFS in-kernel and using zero-copy network buffer techniques would only enhance MBFS's performance. The MBFS server runs on both Solaris and Linux. An MBFS server runs on each machine in the system and implements the



(a) Run-time scalability

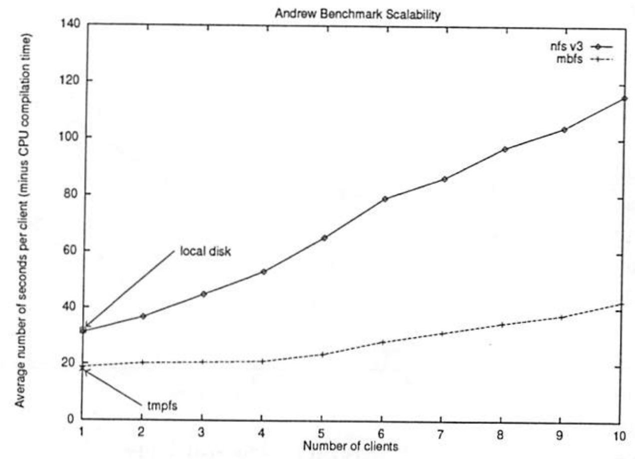


(b) Operation Improvement

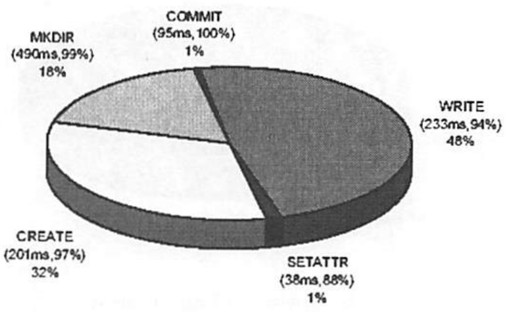
Figure 1: Results of the large write throughput test.

LCM, and DA_1 storage levels. The LCM component monitors the idle resources, accepts LCM storage/retrieval requests, and migrates data to/from other servers as described in [14]. Similarly, the DA_1 component uses local disks for stable storage and employs an addressing algorithm similar to that used by the LCM. To eliminate the improvements resulting from multiple servers (parallelism) and instead focus on improvements caused by variable persistence, we only ran a single server when comparing to NFS.

The MBFS client is implemented as a new file system type at the UNIX VFS layer. Solaris and Linux implementations currently exist. MBFS clients redirect VFS operations to the LCM system or service them from the local cache. The system currently uses the filename-matching interface. The current implementation does not yet support callbacks, so the time_till_next_level of LM must be 0 so data is flushed to the LCM to ensure consistency. The sys-

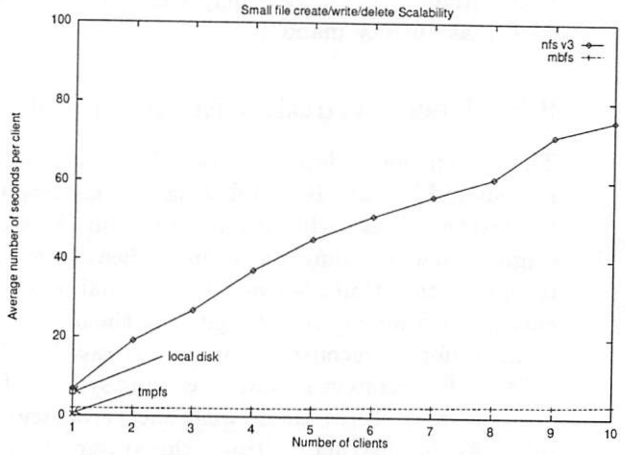


(a) Run-time scalability

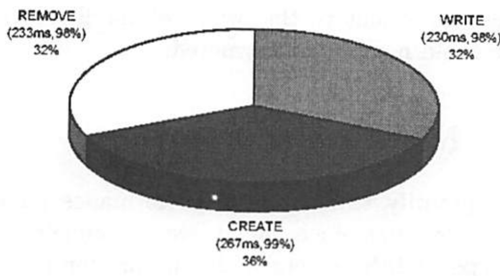


(b) Operation Improvement

Figure 2: Results of the Andrew Benchmark.



(a) Run-time scalability



(b) Operation Improvement

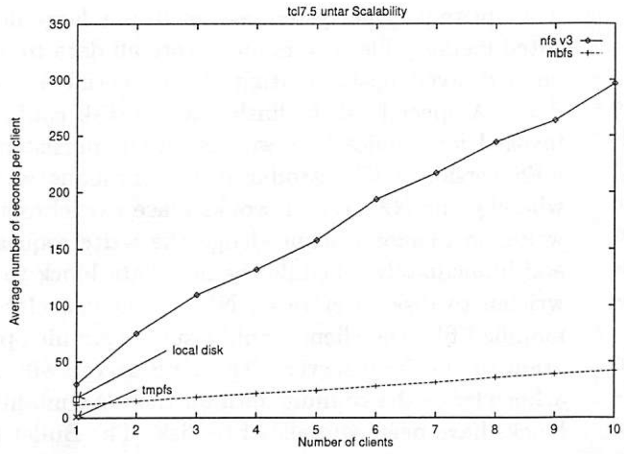
Figure 3: Results of the small file create/write/delete test.

tem currently provides the same consistency guarantees as NFS. Callbacks would improve the MBFS results shown here because file data could stay in the LM without going over the network. Replication is not currently supported by the servers. Communication with the LCM is via UDP using a simple request-reply protocol with timeouts and retransmissions.

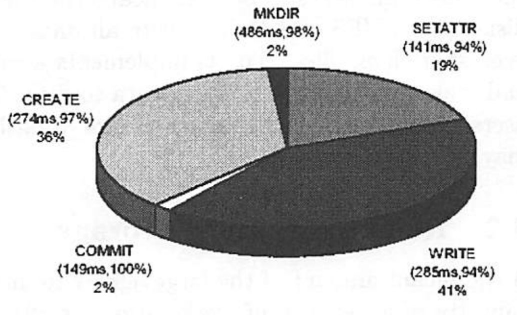
7.1 Solaris Results

The MBFS and NFS servers were run on a SPARC 20 with 128 MB of memory and a Seagate ST31200W disk with a 10ms average access time. We ran up to 10 simultaneous client machines on each server. Each client was a SPARC 20 with 64 MB of memory and a 10ms Seagate local disk (for the UFS tests). Tmpfs tests used the standard UNIX temp file system. All machines were connected by a 100 Mbps Ethernet and tests were run during the evening hours when the network was lightly loaded. We computed confidence intervals for each test. The confidence intervals were typically within 5% of the total runtime and as a result are not shown in the graphs.

Five tests with different workloads were run: (1) write throughput for large files, (2) small file create/write/delete throughput, (3) a mixture of file and directory creation with large and small files, (4) a manual edit test, and (5) the Andrew Benchmark. Each of the tests focused on write traffic to files or directories. Read traffic was minimal and did not contribute to any speedups observed. Each test was run several times and averaged. The results show two graphs: (1) a line-graph illustrating the scalability of MBFS versus NFS in terms of total runtime, and (2) a pie-chart of the 10-client test describing how much each operation contributed to the overall speedup. Each slice of the pie-chart depicts the percentage of runtime improvement caused by that operation. The



(a) Run-time scalability



(b) Operation Improvement

Figure 4: Results of the untar test.

numbers in parenthesis list the average speedup over NSF for the operation. The first number in the pair gives the absolutes speedup in milliseconds and the second number gives the relative speedup in terms of a percentage $(\frac{NFStime-MBFStime}{NFStime})$.

To measure the baseline write performance, we used a large-write throughput test. Each client creates a copy of an existing 10 MB file. Because the new file is a copy, disk persistence is not required. The original 10 MB files is preloaded into the client cache to eliminate read traffic from the test. Figure 1(a) shows that MBFS performs better than NFS (despite NFS's use of asynchronous writes) because of contention at the disk. Note that as the number of clients increases, the server's maximum receive rate quickly becomes the bottleneck in this test. Figure 1(b) shows that 91% of the overall runtime savings were due to improved write operations, with 9% of the improvement arising from the fact the MBFS does not issue a final commit operation. In other words, even when writes are asynchronous, the server response time is significantly slower than MBFS memory-only writes.

Figure 2(a) shows the results of the Andrew Benchmark that tests several aspects of file system performance: making directories, copying files to those directories, scanning the directories, read the file contents, and perform a compilation. To more accurately isolate the file system performance, we subtracted the CPU time used by the compiler during the compilation phase. Because all the data and directories are generated or copied, none of the writes required disk persistence. Improvements range from 40% with one client (a perceivable improvement to the user) to as much as 64%. Figure 2(b) clearly illustrates that all savings come from operations that typically require disk persistence: mkdir, create, write, setattr, and commit.

Figure 3(a) shows the results of the small file test where each client repeatedly (100 times) creates a file, writes 1K of data, and deletes the file. The test measures directory and metadata performance and models applications that generate several small files and then deletes them (for example, compilers). The results are extremely impressive with MBFS processing 313 files per second compared with NFS's 13 per second at 10 clients.

Figures 4(a) and 4(b) show the results of untaring the TCL 7.5 source code. Untar is an I/O bound program that creates multiple directories and writes a variety of file sizes (all easily recreatable from the tar file). Again, the results are perceivably faster to the user.

In all the tests MBFS outperformed both NFS and UFS (local disks). More surprising is how often MBFS challenged tmpfs performance despite the lack of callbacks in the current implementation. Similar performance tests were performed in a Linux environment with even better results since Linux's NFS version 2 does not support asynchronous writes.

Finally, we ran an edit test in which we edited various files, composed email messages, and created web pages. All files required disk persistence. As expected there was no, or minimal, performance gains.

8 Related Work

Several researchers have investigate high-speed persistent storage techniques to improve write performance. Remote memory systems have also been proposed as a high-speed file caches. The following briefly describes this related work.

8.1 Persistence vs. Performance

The performance penalties of a disk-persistent file storage model are well known and have been addressed by several file systems [3, 18, 6, 9, 5, 25, 20]. Unlike the application-aware persistence design we propose, the following systems have attempted to improve performance without changing the conventional one-size-fits-all disk-persistence file system abstraction.

The xFS file system [3] attempts to improve write performance by distributing data storage across multiple server disks (i.e., a software RAID [22]) using log-based striping similar to Zebra [16]. To maintain disk-persistence guarantees, all data is written to the distributed disk storage system. xFS uses metadata managers to provide scalable access to metadata and maintain cache consistency. This approach is particularly useful for large writes but does little for small writes.

The Harp[18] and RIO[6] file systems take an approach similar to the one used by Derby. High-speed disk persistence is provided by outfitting machines with UPS's to provide non-volatile memory storage. Harp also supported replication to improve availability. However, Harp used dedicated servers as opposed to Derby which uses the idle memory of any machine. RIO uses non-volatile memory to recover from machine failures and could be used to implement the non-volatile storage of Harp or Derby. Alternatively it could be used to implement memory-based writes on an NFS file server, but would not take advantage of the idle aggregate remote memory storage space like DERBY. Also, because RIO does not flush data to disk (unless memory is exhausted), UPS failures may result in large data losses. Because Derby only uses UPS's as temporary persistent storage, UPS failures are less catastrophic.

Other systems have introduced the concept of delayed writes (asynchronous writes) to remove the disk from the critical path. For example, conventional Unix file systems use a 30-second delayed-write policy to improve write performance but create the potential for lost data. Similarly, Cortes et al. [9] describe a distributed file system called PAFS that uses a cooperative cache and has cache servers and disk servers. To remove disks from the critical path, PAFS performs aggressive prefetching and immediately acknowledges file modifications once they are stored in the memory of a cache server. Cache servers use a UNIX-like 30 second delayed write policy at which point they send the data to a disk server. The Sprite [20] file system assumed very large dedicated memory file servers and wrote all data to disk on a delayed basis creating the potential for lost data. A special call to flush data to disk could be invoked for applications worried about persistence. NFS version 3 [5] introduced asynchronous writes whereby the NFS server would place asynchronous writes in memory, acknowledge the write requests, and immediately schedule the new data block to be written to disk. Before an NFS client can close a modified file, the client would issue a commit operation to the NFS server. The NFS server will not acknowledge the commit until all the file's modified blocks have been committed to disk. The Bullet file server [25] provides a "Paranoia Factor" which when set to zero provides the equivalent of asynchronous writes. For other values, N, of the paranoia factor, the Bullet file server would replicate the file on N disks. Both NFS and Bullet write all data to disk, even short lived files. Tmpfs implements a ramdisk and makes no attempt to write data to disk. Tmpfs users understand that tmpfs files are volatile and may be lost at any time.

8.2 Remote Memory Storage

A significant amount of the large aggregate memory capacity of a network of workstations is often idle. Off-the-shelf systems provide access to this idle remote memory an order of magnitude faster than disk latencies. Therefore, many systems have been developed to make use of idle memory capacity, primarily for paging and caching.

Comer and Griffioen [7] introduced the *remote memory model* in which client machines that exhaust their local memory capacity paged to one of a set of dedicated *remote memory servers.* Each client's memory was private and inaccessible even if it was idle. Data migration between servers was not supported.

Felten and Zahorjan [11] enhanced the remote memory model to use any idle machine. Idle client machines advertise their available memory to a centralized registry. Active clients randomly chose one idle machine to page to. Like [7], data was not migrated among idle clients.

Dahlin et al. [10] describe an N-Chance Forwarding algorithm for a cooperative cache in which the file caches of many client machines are coordinated to form a global cache. N-Chance Forwarding tries to keep as many different blocks in global memory as it can by showing a preference for *singlets* (single copies of a block) over multiple copies. The cache only stores *clean* (unmodified) blocks. Thus, all file block modifications are written to the file server's disk. A similar approach is used in xFS[3] and PAFS[9].

Feeley et al. [13] describe the Global Memory Service (GMS) system that integrates global memory management at the lowest level of the operating system enabling all system and higher-level software, including VM and the file cache, to make use of global memory. GMS uses per node page age information to approximate global LRU on a cooperative cache. Like N-Chance Forwarding, GMS's only stores *clean* file blocks and so all file writes must hit the file server's disk.

Hartman and Sarkar [23] present a *hint-based* cooperative caching algorithm. Previous work such as N-Chance Forwarding [10] and GMS [13] maintain *facts* about the location of each block in the cooperative cache. Although block location hints may be incorrect, the low overhead needed to maintain hints outweighs the costs of recovering from incorrect hints. All file modifications are written to the file server's disk so that if a hint is missing or incorrect, a client can always retrieve a block from the server. Using hints, block migration is done in a manner similar to that of GMS [13]. Unlike MBFS, none of the above systems considers a client's CPU or memory load when deciding the movement or replacement of pages.

Franklin et al. [12] use remote memory to cache distributed database records and move data around using an algorithm similar in nature to that of N-chance forwarding. Client load was not considered by the data migration mechanism.

The Trapeze network interface [2] provides an additional order of magnitude improvement in remote memory latencies versus disk latencies by improving the network subsystem.

8.3 Application Aware Storage

The Scalable I/O Initiative (SIO), introduced a new file system interface [8] for parallel file systems. The interface enables.direct client control over client caching algorithms, file access hints, and disk layout. As such, it is a suitable platform for integrating our proposed variable persistence model.

9 Summary

In this paper we presented a new file system abstraction for memory-based file management systems. The abstraction is unique in the fact that it allows applications or users to control file persistence on a per-file basis. Applications that can tolerate weak persistence can achieve substantial performance improvements by selecting memory-based storage. The new abstraction has two major extensions to conventional file systems abstractions. First, applications can define a file's persistence requirements. Second, applications can specify rules to reconstruct file data in the unlikely event that it is lost. Analysis of current file systems indicates that a large percentage of write traffic can benefit from weak persistence. To support large amounts of data with weak persistence guarantees, we developed a high-speed loosely-coupled memory storage system that utilizes the idle memory in a network of workstations. Our prototype implementation of the MBFS system running on Solaris and Linux systems shows applications speedups of an order of magnitude or more.

10 Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. We would also like to thank our shepherd, Darrell Long, for his insightful suggestions and comments. Also thanks to the CS570 class for using the system during the fall 1998 semester.

References

[1] A. Acharya, G. Edjlali, and J. Saltz. The Utility of Exploiting Idle Workstations for Parallel Computation. In *Proceedings of 1997 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Seattle, June 1997.

[2] D. C. Anderson, J. S. Chase, S. Gadde, A. J. Gallatin, K. G. Yocum, and M. J. Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 143–154, Berkeley, USA, June 15–19 1998. USENIX Association.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[4] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212. Association for Computing Machinery SIGOPS, October 1991.

[5] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification, June 1995. RFC 1813.

[6] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating systems crashes. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, Cambridge, Massachusetts, 1–5 Oct. 1996. ACM Press.

[7] D. Comer and J. Griffioen. A New Design for Distributed Systems: The Remote Memory Model. In *The Proceedings of the 1990 Summer USENIX Conference*, pages 127–136. USENIX Association, June 1990.

[8] P. Corbett, J. Prost, J. Zelenka, C. Demetriou, E. Riedel, G. Gibson, E. Felten, K. Li, Y. Chen, L. Peterson, J. Hartman, B. Bershad, and A. Wolman. Proposal for a Common Parallel File System Programming Interface Part I Version 0.62, August 1996.

[9] T. Cortes, S. Girona, and J. Labarta. Avoiding the cache-coherence problem in a parallel/distributed file system. In *Proceedings of the High-Performace Computing and Networking*, pages 860–869, Apr. 1997.

[10] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative Caching: Using Remove Client Memory to Improve File System Performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[11] E. W. Felten and J. Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical Report 91-03-09, Department of Computer Science and Engineering, University of Washington, March 1991.

[12] M. Franklin, M. Carey, and M. Livny. Global Memory Management in Client-Server DBMS Architectures. In *18th International Conference on Very Large Data Bases*, 1992.

[13] M. Freeley, W. Morgan, F. Pighin, A.Karlin, and H. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proceedins of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[14] J. Griffioen, T. Anderson, and Y. Breitbart. A Dynamic Migration Algorithm for a Distributed Memory-Based File Management System. In *Proceedings of the IEEE 7th International Workshop on Research Issues in Data Engineering*, April 1997.

[15] J. Griffioen, R. Vingralek, T. Anderson, and Y. Breitbart. Derby: A Memory Management System for Distributed Main Memory Databases. In *The Proceedings of the IEEE 6th International Workshop on Research Issues in Data Engineering (RIDE '96)*, February 1996.

[16] J. Hartman and J. Ousterhout. Zebra: A Striped Network File System. In *Proceedings of the Usenix File System Workshop*, pages 71–78, May 1992.

[17] H. V. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dali: A High Performance Main Memory Storage Manager. In *Proceedings of the VLDB Conference*, 1994.

[18] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 226–38. Association for Computing Machinery SIGOPS, October 1991.

[19] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith. Andrew: A Distributed Personal Computing Environment. *CACM*, 29:184–201, March 1986.

[20] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, Feb. 1988.

[21] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[22] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *ACM SIGMOD 88*, pages 109–116, June 1988.

[23] P. Sarkar and J. Hartman. Efficient Cooperative Caching using Hints. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 35–46, October 1996.

[24] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In B. Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 146–160, New York, NY, USA, Dec. 1993. ACM Press.

[25] R. van Renesse, A. S. Tanenbaum, and A. Wilschut. The Design of a High Performance File Server. *Proceedings of the IEEE 9th International Conference on Distributed Computing Systems*, 1989.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight–ten times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and C++, book and software reviews, summaries of sessions at USENIX conferences, and Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT – as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Discount on BSDI, Inc. products.
- Discount on all publications and software from Prime Time Freeware.
- Savings (10-20%) on selected titles from Academic Press, Morgan Kaufmann, New Riders/Cisco Press/MTP, O'Reilly & Associates, OnWord Press, The Open Group, Sage Science Press, and Wiley Computer Publishing.
- Special subscription rate for *The Linux Journal, The Perl Journal, IEEE Concurrency,* and all Sage Science Press journals.

Supporting Members of the USENIX Association:

C++ Users Journal
Cirrus Technologies
Cisco Systems, Inc.
CyberSource Corporation
Deer Run Associates
Greenberg News Networks/MedCast Networks
Hewlett-Packard India Software Operations
Internet Security Systems, Inc.
Microsoft Research
MKS, Inc.
Motorola Australia Software Centre
NeoSoft, Inc.
New Riders Press
Nimrod AS
O'Reilly & Associates Inc.
Performance Computing
Questra Consulting
Sendmail, Inc.
Server/Workstation Expert
TeamQuest Corporation
UUNET Technologies, Inc.
Windows NT Systems Magazine
WITSEC, Inc.

Sage Supporting Members:

Atlantic Systems Group
Collective Technologies
D. E. Shaw & Co.
Deer Run Associates
Electric Lightware
ESM Services, Inc.
GNAC, Inc.
Mentor Graphics Corp.
Microsoft Research
MindSource Software Engineers
Motorola Australia Software Centre
New Riders Press
O'Reilly & Associates Inc.
Remedy Corporation
SysAdmin Magazine
Taos Mountain
TransQuest Technologies, Inc.
Unix Guru Universe (UGU)

For further information about membership, conferences or publications, contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.
Phone: 510-528-8649. Fax: 510-548-5738.
Email: *office@usenix.org.*
URL: *http://www.usenix.org.*

ISBN 1-880446-33-2